THE UNIVERSITY OF MICHIGAN

Technical Report 7

CONCOMP

June 1968

THE SYNTACTIC STRUCTURE OF MAD/1

David L. Mills

THE UNIVERSITY OF MICHIGAN

Technical Report 7

THE SYNTACTIC STRUCTURE OF MAD/I

David L. Mills

CONCOMP: Research in Conversational Use of Computers
F.H. Westervelt, Project Director
ORA Project 07449

supported by:

ADVANCED RESEARCH PROJECTS AGENCY
DEPARTMENT OF DEFENSE
WASHINGTON, D.C.

CONTRACT NO. DA-49-083 OSA-3050
ARPA ORDER NO. 716

administered through

OFFICE OF RESEARCH ADMINISTRATION ANN ARBOR

June 1968

TABLE OF CONTENTS

LIST OF FIGURES

BLANK PAGE

THE SYNTACTIC STRUCTURE OF MAD/I

I. INTRODUCTION

The various dialects of MAD developed at The University of Michigan and elsewhere can be described as ALGOL-like languages with strong flavors of FORTRAN. The language has enjoyed considerable popularity at the University in both teaching and research during a developmental evolution which began in 1960 with an IBM 704 version of the compiler and progressed to the present IBM 7090 version. The MAD language itself is designed to be readily taught to relatively unsophisticated students and yet to provide the power of generality of expresssion necessary in sophisticated research applications. In general, the compiler implementations have been finely tuned for high-speed translation and for production of reasonably good object code. The list of references at the end of this report contains a compendium of reference material covering the development of the compiler and the structure of the language. In the subsequent discussion of this report a working familiarity with the MAD language will be assumed in programming examples, although this is not strictly necessary for an understanding of the principles involved.

1.1 Evolution of MAD/I

In mid-1965 the University began a gradual systems change-over from the IBM 7090 to the System/360 Model 67. The development of the System/360 system was predicated upon the virtual-memory concept, which involves a hardware-assisted

dynamic address translation procedure in which each concurrent system program is written as if it owned all the addressable core storage of the machine. Successful operation of this procedure requires a high-speed backup storage, such as a drum, for temporary storage of core memory overflows and furthermore a reliance upon a sharable system program structure. The implementation of the Michigan Timesharing System (MTS) is based on these concepts and represents the environment in which both the new MAD compiler and its compiled programs will operate.

At its inception the MAD project was faced with two alternative developmental paths. On the one hand a MAD translator could be implemented for the Model 67 which would be a virtual transliteration of the existing MAD/7090 translator and with few additional features. On the other hand a new language could be developed which contained all those useful features of the existing MAD/7090 translator and in addition many new ones required for such applications as the development of graphics languages. The former effort would at least provide a continuance of the MAD/7090 language itself, a factor thought vital in the almost captive MAD-committed user population. The latter effort would be expected to provide, in addition to the valuable developmental experience itself, a sound theoretical framework bolstering the specification of a new language called MAD/I and the construction of its compiler. In addition, the framework developed would include a systematic procedure for the specification of new language families, based on MAD/I, within

which specialized languages suitable for the manipulation of data structures could be developed.

Although, in the beginning, the developmental effort of the MAD project was concentrated along the former or transliteration path, a gradual shift in emphasis took place, to such an extent that the developmental effort at this time is almost solely concentrated in the specification of MAD/I and the implementation of its compiler. The new language is in many respects very much like the old. For instance, the assignment, transfer, conditional, iteration, and input/output statements are incorporated into the MAD/I language in substantially the same way as into MAD/7090. Variables, constants, functions, arrays, and expressions have the same interpretation in both languages. Several minor differences exist between the two languages, however, in the rules for the naming of statements, the scope of compound statements, and the elements of input/output statements.

The major differences between the two languages occur in the inclusion of comprehensive definitional facilities and the introduction of new data structural types. In the MAD/7090 language a definitional facility was implemented which provided for the introduction of new data types and for the definition of a restricted class of operations upon them. In the new new language this facility has been expanded so that, not only a much richer class of data types can be defined, but quite general operations can be performed upon them. In order to implement this expanded definitional facility, a new

metalanguage has been developed in which the definitions are expressed. In fact, all of the MAD/I statements announced in the programming manuals have been implemented in this new metalanguage.

The impact of the systematic introduction of new data types is most obvious in the syntactic specification of the MAD/I declaration statements. Although the MAD/7090 concepts of dimension, storage mapping, and mode have validity in MAD/I programs, their interpretation is far more general. For instance arrays may contain arrays as elements, and the storage assigned to them may vary dynamically during execution and be shared among several functions. Linkages between functions are far more flexible, and dynamic loading and overlay operations are possible. In short, the declaration features of the language allow maximum advantage to be taken of the virtual-memory concept and the time-sharing environment in which MAD/I programs are executed.

The broadening of scope and generality as compared with MAD/7090 has not been achieved without a corresponding loss of compatibility in respect to the older language. In fact, the characteristics of the 7090 as compared to those of the Model 67 seem to prejudice a virtue of compatibility in the first place. As a result, many common gimmicks popular in MAD/7090 programming simply have no counterpart in MAD/I programming. However, the converse most certainly will be far more likely, in spite of the fact that old programming habits die hard. The most common incompatibilities are of course related to the character set

and the byte addressing structure of the Model 67, and this directly affects those operations of bitwise shifting and masking of data, and the resolution of storage addresses. A translator has been constructed to aid in the conversion of MAD/7090 programs to their MAD/I counterparts, and has proved useful in the majority of cases. In some cases involving packing/unpacking and character-sensitive operations, translation is not possible unless a highly sophisticated processor is postulated. Unfortunately, the MAD language has been particularly convenient in the construction of symbol manipulation programs; and a large body of extant and useful programs are unavoidably threatened with obsolescence as a result of the eventual change-over to MAD/I.

As a consequence of the power inherent in the definitional facilities of the compiler, it is apparent that a description of the language in terms of its syntax would be misleading at best. Obviously the structure of the translator provides the capabilities for the definition of a rather wide class of languages, each one characterized by a consistent set of statements of the definitional metalanguage. One of these sets of definitional statements just happens to represent the language called MAD/I in the programming manuals, but any other consistent set of definitional statements might have been chosen as well. The MAD/I set was chosen rather arbitrarily to represent that language thought most useful and economical for the widest class of potential users, yet with a large capability

for enrichment through the inclusion of special-purpose definitional packages.

The most useful description of the MAD/I language and its translator then demonstrably involves the syntactic specification of those constructs which can be identified by the various analysis algorithms embedded within the translator and a description of the operations possible upon these constructs. These tasks will dominate the discussion for the remainder of this report. However, many examples drawn from the MAD/I language will be used from time to time to explicate the discussion.

It should be noted that the procedures described herein used to analyze the syntactic specification of MAD and to construct its compiler are applicable to other than algebraic-type languages. In fact, the same analysis techniques have been used in the construction of a machine-language assembler and in the specification of a computer-to-computer message transmission protocol.

II. FORMAL SYNTACTIC SPECIFICATION

The formal linguistic structure which describes the MAD syntax can be described as a modified operator precedence grammar. This structural description provides an exceptionally sound framework which satisfies both the needs of syntactic flexibility in the definition of statement forms and of structural integrity in the control of error recovery. The approach

taken in the formulatory steps of the formal syntactic specification is first to construct a kernel language of the operator precedence type and then to construct a set of context-dependent transformations which operate upon sentences of the source language to produce sentences of the kernel language. Since it is known that the family of precedence languages are unambiguous and have rather good error-recovery characteristics; then, if the context-dependent transformations are carefully chosen, the resultant language should be considerably richer than the operator-precedence kernel language and yet retain many of its desirable characteristics.

2.1 Terminology

A terminal vocabulary V_T is a set of symbols chosen as the alphabet of the language. A language L is a collection of certain strings of all those strings formed by indefinite concatentions of elements of V_T. Each of these strings is a sentence S of L and is generated by applications of a set of rules called a grammar G. In the grammars discussed here each of these rules or productions take the form $U \rightarrow x$, where U is an element of a nonterminal vocabulary V_N and x is a string over $V_T + V_N$, called simply the vocabulary. Furthermore, every S in L is assigned a structural description by G which demonstrates how that string is decomposed into its constituent structural units, each labeled by an element of V_N.

The productions of G thus form an effective procedure for deciding whether any string over the vocabulary is or is not a sentence of the language. Furthermore, since every meaningful constituent substring or prime phrase of a sentence is assigned a nonterminal symbol by a production of G , then the identification of a prime phrase during the decision process can be made synonymous with the production of some arbitrary interpretation or translation of the elements of the prime phrase itself.

If all productions of G take the form $U \rightarrow x$ as above, then L is described as context-free and the decomposition or parsing of a sentence into its constituent structural units involves relatively simple techniques. On the other hand, if some of the productions are of the form $xUy \rightarrow z$, where x , y, and z are strings over V , then L is described as context-dependent, and more complicated parsing techniques are required. A production-oriented description of MAD/I is necessarily context-dependent, although by far the majority of productions are of the context-free type.

Any useful programming language like MAD/I should be capable of being described by a particular grammar in such a way that each sentence of the language is assigned exactly one structural description, or, equivalently, that only one parse exists. If such is the case, then the language is described as unambiguous. Although it is not in general possible

to determine whether a particular phrase-structure grammar is or is not unambiguous, certain families of phrase-structure grammars can be shown to have this property. One of the most useful of these families is that of the precedence grammars; and, of these, the operator precedence grammars are particularly suited to the description of MAD/I. In fact it is convenient to describe the bulk of MAD's syntax in an operator-precedence grammar and then to describe those few exceptions by means of context-dependent transformations which are applied to the source text prior to the operator-grammar parsing algorithm.

There is one significant problem connected with this approach. The useful operator precedence grammar parsing techniques operate upon the terminal symbols of a sentence producing progressively larger prime phrases as intermediate parses and finally terminating when the entire sentence has been scanned. Such a process, commonly called a bottom-up parse, is highly adaptable to the parsing of the lower-level algebraic expression structures in the language. On the other hand, the parsing of the higher-level statement structures is intuitively a much more goal-oriented process, and a more general top-down process is needed. In the syntactic specification of MAD/I, the productions are carefully chosen so that contextual features can provide clues for a macro-driven top-down statement scan, yet retain the advantages of a bottom-up expression scan.

2.2 Production Systems

The set of productions defining a grammar may be represented in any of several common notational schemes, the most common of which may be the Backus Normal Form (BNF). The particular notational scheme followed herein is an adaptation of the BNF and is defined as follows:

Each production P consists of a left part U, which is a particular symbol of V_N, and a right part x, which is a string over $V = V_T + V_N$. In general there may be more than one production with the same left part, each such production corresponding to an instance of a component in a BNF rule. It will be assumed that no right part is the null string, for it can be shown that a grammar containing a production with a null right part can be naturally rewritten without such a production and without materially affecting the generative capacity of the grammar.

A grammar, each rule of which takes one of the following forms:

$$U \rightarrow a \qquad 1.$$

$$U_1 \rightarrow aU_2 \qquad 2.$$

$$U_1 \rightarrow U_2 a \qquad 3.$$

$$U_1 \rightarrow aU_2 b, \qquad 4.$$

where U_i are elements of V_N and a, b are strings over V_T, is called a linear grammar. These grammars are characterized by the fact that, in each production, only a single nonterminal

symbol on the right side is replaced or rewritten by the nonterminal on the left; and, furthermore, each such rewrite (except those corresponding to Rule 1) has fewer symbols than the previous. If we add to these four forms the following

$$U_1 \rightarrow U_2 \qquad 5.$$

$$U_1 \rightarrow U_2 a U_3 \qquad 6.,$$

and require a and b to be single elements of V_T, then an appropriate paradigm for an algebraic language production system is evident. Here the terminal symbol a in Rule 1 corresponds to the notion of operand, and the terminal symbols in the remaining rules correspond to the notion of operator. The nonterminal symbols correspond to the notions of expression and statement, depending upon the hierarchy of the production system.

Note that these six rules represent all of the production forms of an operator grammar (see below) which have right sides of lengths no greater than three and, furthermore, contain no sequences of two or more contiguous terminal symbols. Although sequences of this type can occur in an operator grammar, nevertheless, each such sequence can be mapped into a single element of a set of metaterminals for convenience, and this practice will be followed henceforth.

Rule 1 establishes a duality between the notion of operand and that of nonterminal symbol. In general, in an algebraic language grammar there is a derivation or sequence of applications of the rules of the grammar starting with each

and every nonterminal symbol of the grammar and ending with an operand. Using the notion of metaterminal mentioned above, it is clear that only a single Rule 1 is necessary in an algebraic language grammar. Rules 2 and 3 represent the types of productions associated with the unary prefix and unary postfix operators in the language, and Rule 6 represents the type of production associated with the binary operators. Rule 4 represents the type of production associated with parenthesized groupings, and Rule 5 represents really only a notational convenience so that the grammar can be expressed in a more compact form.

By convention, each production whose form coincides with Rules 2, 3, and 6 above will be identified by its single terminal symbol, which serves as a referent in the application of the semantic interpretation rules or macro transformation associated with the production. Thus, when a prime phrase is identified by the bottom-up parsing algorithm, it is only necessary to identify whether its form coincides with Rule 2, 3, or 6 and which operator is involved. The nonterminal symbols of the prime phrase play no part in this determination.

2.3 Operator Precedence Grammars

A particular grammar can be found to belong to the family of precedence grammars by application of a certain technique which results in the assignment of one or more binary relations between each pair of symbols of the vocabulary $V = V_N + V_T$. These relations can be symbolized as $\circ$ (null)

$<\cdot$, $\doteq$, and $\cdot>$, and summarized in an n x n matrix, where n is the number of symbols of V . If no more than one of these four relations holds between any such pair in the language, then the grammar belongs to the class of simple precedence grammars. The precedence matrix so constructed can serve as the driving table in a simple algorithm which decomposes a sentence of the language into its prime phrases.

The sheer size of the precedence matrix for a language of some complexity (146x146 for the MAD/I case) encourages further restriction in the grammar to exclude those productions which contain adjacent nonterminal symbols. Such grammars, known as the operator precedence grammars, are characterized by a m x m precedence matrix, where m is the number of symbols of V_T . A good deal of violence is done to some natural syntatic descriptions when this restriction is enforced, although several techniques are available to enrich such a language by the introduction of metaterminal symbols consisting of certain strings over $V_N + V_T$. A certain rationale is available, then, to restrict the kernel structural description of MAD/I to an operator precedence grammar.

A verification procedure, due to Floyd (see References), is available with which it is possible to determine whether or not a particular operator grammar is a member of the precedence family or not. The procedure can be implemented either recursively or iteratively as a computer program. Both techniques have been implemented as MAD/7090 programs, with the

latter technique enjoying a speed advantage of about ten-to-one over the former. The latter technique can be illustrated by the algorithms described below. In the following, U represents an element of V_N and T an element of V_T. A string over $V = V_T + V_N$ is represented by a lower-case letter.

The process of constructing a precedence matrix for an operator grammar consists of two steps: In the first step, two tables are constructed showing for each nonterminal symbol $U \in V_N$ those terminal symbols which can occur as the leftmost and rightmost symbols respectively in a derivation of U. The table of leftmost terminal derivatives (LTD) can be constructed by the following process:

1. For each production $U_1 \rightarrow T_1 x$ or $U_1 \rightarrow U_2 T_1 x$, enter T_1 as an LTD of U_1.

2. For each production $U_1 \rightarrow U_2 x$, enter every LTD of U_2 as an LTD of U_1.

3. Repeat step 2 until, in a finite number of steps, the process converges.

The table of rightmost terminal derivatives (RTD) is constructed in the analogous way.

The second step for constructing the precedence matrix for an operator grammar involves the two LTD and RTD tables just constructed, the algorithm below, and the precedence matrix itself, an $n \times n$ square matrix where n is the number of symbols of V_T. The algorithm cited assigns four relations,

one or more of which must hold between two terminal symbols T_1 and T_2 :

1. $T_1 \doteq T_2$ if there is a production $U \rightarrow xT_1T_2y$ or $U \rightarrow xT_1U_1T_2y$.

2. $T_1 \lessdot T_2$ if there is a production $U \rightarrow xU_1T_2y$ and T_1 is an RTD of U_1 .

3. $T_1 \gtrdot T_2$ if there is a production $U \rightarrow xT_1U_1y$ and T_2 is an LTD of U_1 .

4. $T_1 \circ T_2$ if none of the above holds.

If no more than one of these relations holds between any two terminal symbols T_1 and T_2 , then the operator grammar is in fact an operator precedence grammar. Note that if T_1 and T_2 were not constrained to be elements of V_T , but could in fact be elements of $V_N + V_T$, then the same process would result in a precedence matrix for a simple precedence grammar.

Figure 1 summarizes those steps in the construction of the precedence matrix for a simple algebraic-like language taken from Floyd (see References). The equivalent steps for the derivation of the precedence matrix for MAD/1 are summarized in Section 4.1. In this and subsequent examples the metaterminal symbols will be assigned in each instance as names prefixed by percent signs (%) . In this figure the void $\circ$ relation is assumed to hold in all those positions of the matrix in which a blank is evident Blank positions in the

Productions

S→A

A→A + B

A→B

B→B * C

B→C

C→(A)

C→%I

Nonterminal Vocabulary

S A B C

Terminal Vocabulary

+ * ( ) %I

Left Terminal Derivatives

NTC Terminal Characters

S	+	*	(	%I
A	+	*	(	%I
B	*	(	%I	
C	(	%I		

Right Terminal Derivatives

NTC Terminal Characters

S	+	*	)	%I
A	+	*	)	%I
B	*	)	%I	
C	)	%I		

Precedence Matrix

	+	*	(	)	%I
+	·>	⋖	⋖	·>	⋖
*	·>	·>	⋖	·>	⋖
(	⋖	⋖	⋖	≐	⋖
)	·>	·>		·>	
%I	·>	·>		·>	

Figure 1. Floyd's Simple Grammar.

matrix correspond to those cases where a void precedence relation exists and provide either an opportunity for a context-dependent transformation or an indication of an incorrect program, that is, an occurrence of a sentence not in the language.

It is possible, reputably in all useful cases and certainly here, to represent the nonvoid three precedence relations between any two terminal symbols in a conveniently compact form which assigns two integers to every terminal symbol. These integers might be called the left and right precedence functions and represent the "order" precedence relation in the same fashion as the matrix when the left function of the leftmost symbol is compared to the right function of the rightmost symbol in a true order relation. Both of these precedence functions are shown for Floyd's simple grammar in Table 1. It is possible

Terminal Character	Precedence Functions	
	F	G
+	3	2
*	5	4
(	1	6
)	5	1
%I	5	6

Table 1.

in some cases to dispense with one of these functions and to represent the precedence relations as a single integer assigned to each terminal symbol, as is done in fact in 7090 MAD. The

generality of the new MAD/I does not evidently permit this simplification (see Section 4.1).

In practice it has not been necessary to represent the entire precedence matrix for MAD/I within the compiler, but only a much smaller matrix which shows whether or not a nonvoid precedence relation exists between any two terminal symbols. The internal descriptor corresponding to each nonterminal symbol in the language has coded within it an index into this compact matrix as well as both the left and right precedence functions. This compact matrix, called the terminal context matrix, has importance in other uses and is discussed further below.

2.4 Contextual Features

If the grammar for a practical algebraic language could be made as simple as Floyd's example presented in the previous section, then the parsing algorithm could be exceptionally simple; indeed, Floyd gives an example of such an algorithm. In the more complex practical cases, a good deal of contextual information must be available to provide handles for such context-dependent transformations as those to resolve the syntax of binary operators used in unary contexts and so forth. The discussion in this section will be concerned with the development of certain tables and matrices which are highly useful in gaining insight into the contextual structure of the language generated by a context-free grammar. As implied, the development of these tables and matrices does not require that the grammar be an operator or a precedence grammar.

The allowable pairs of terminal symbols in the language generated by a context-free grammar can be determined with the following two-step procedure (due to Floyd). The results are summarized in an $m \times m$ terminal matrix, where m is the number of symbols of V_T. The procedure is similar in nature to that outlined above for the construction of the precedence matrix. In the first step two tables are constructed, each giving respectively the left most and rightmost symbols of $V = V_N + V_T$ which may occur in a derivation for a nonterminal symbol. The table of leftmost symbols (LS) is constructed by the following process:

1. For each $X \in V$, enter X as an LS of X.

2. For each production $U \rightarrow Xy$, enter each LS of X as an LS of U.

3. Repeat step 2 until, in a finite number of steps, the process converges.

The table of rightmost symbols (RS) is constructed by an analogous process.

The second step in the construction of the terminal matrix involves consideration of all pairs of adjacent symbols XY which may occur in the right part of a production. If a is a terminal symbol which is an RS of X, and b is a terminal symbol which is an LS of Y, then ab is an allowable terminal symbol pair in the language. The terminal matrix corresponding to Floyd's simple grammar is shown in Figure 2.

	+	*	(	)	%I
+			T		T
*			T		T
(			T		T
)	T	T		T	
%I	T	T		T	

Figure 2. Terminal Matrix.

As before, blank positions in the matrix correspond to invalid constructions and can be used in connection with context-dependent transformations.

During the scan of certain statement types it becomes convenient to invoke the statement-scanning algorithm from a macro transformation (see Section 3.2) at a higher syntactic level. The algorithm is expected to terminate in the identification of one of the nonterminal symbols described in connection with the production system. The contextual features necessary to properly initiate and terminate such a procedure can be summarized in a pair of tables, each giving respectively the left and right terminal symbol delimiters which may bracket the nonterminal symbol to be identified as the goal of the procedure. The algorithm is given below.

The table of left terminal symbol delimiters (LSD) can be constructed by the following process:

1. For every production $U \rightarrow xT_1U_1y$ enter T_1 as an LSD of U_1.

2. For every production $U \rightarrow U_1x$ enter every LSD of U_1 as an LSD of U.

3. Repeat Step 2 until in a finite number of steps the process converges.

The table of right terminal symbol delimiters (RSD) is constructed by an analogous process. Figure 3 shows these two tables as derived from Floyd's simple grammar.

Left Terminal Delimiters

NTC	Terminal Characters		
S			
A	(		
B	+	(	
C	+	*	(

Right Terminal Delimiters

NTC	Terminal Characters		
S			
A	+	)	
B	+	*	)
C	+	*	)

Figure 3. Terminal Delimiter Tables.

The most useful of all the various tables and matrices discussed so far is a three-dimensional array called the terminal context matrix. This matrix, used in the application of context-dependent transformations, indicates for every pair of terminal symbols a and b whether:

1. the pair ab is allowable in the language,
2. a nonvoid precedence relation exists between a and b .

The matrix can be considered as two layers of a p x p square array, the ith column and ith row of which are identified by an equivalence class. The equivalence classes are constructed from the precedence and terminal matrices as follows:

1. Construct an m x m square matrix, the ith column and ith row of which are identified by each of the m symbols of V_T . Each element of the matrix is identified by its coordinates as the element of the a_ith row and a_jth column, where a_i and a_j are symbols of V_T . Each such element is a coded number from which can be determined
   a. whether the adjacent symbol pair $a_i a_j$ is allowable,
   b. whether a nonvoid precedence relation exists between a_i and a_j .
2. From this matrix a reduced matrix is constructed by deleting equivalent rows and columns in the following way:

if a_i and a_j identify two rows and in addition the corresponding two columns, then a_i and a_j belong to the same equivalence class if the rows identified by a_i and a_j are identical and in addition the columns identified by a_i and a_j are identical. The resultant matrix will have p rows and p columns.

3. The terminal context matrix is then constructed from the reduced matrix by associating with the first $p \times p$ layer a set of integer-valued elements which, for the a_ith row and a_jth column, take on the value one if $a_i a_j$ is an allowable terminal pair and zero otherwise. The second $p \times p$ layer is constructed in the same manner of the same elements, which take on the value one if a nonvoid precedence relation exists between a_i and a_j and zero otherwise.

The equivalence classes and terminal context matrix derived from Floyd's simple grammar are shown in Figure 4. In this figure the letter T stands for a one in the first layer and the letter P for a one in the second layer. In the construction of the terminal context matrix a partition of V_T has been achieved which assigns to each symbol of V_T a syntactic class number which is an index to a row or column of the terminal context matrix. Each terminal and metaterminal symbol of the MAD/I language is assigned such a syntactic class number along with its left and right precedence functions as part of the internal descriptor developed within the compiler.

Equivalence Classes

CL	Rep	Members
01	+	*
02	(	
03	)	
04	%I	

Matrix

	+	(	)	%I	
+	P	PT	P	PT	01
(	P	PT	P	PT	02
)	PT		PT		03
%I	PT		PT		04
	01	02	03	04	

Figure 4. Terminal Context Matrix.

The motivation for constructing the terminal context matrix in just this manner will become clearer subsequently upon consideration of context-dependent transformations. It may be pointed out here that the elements of each of the two layers may take on values other than zero and one in connection with these transformations, and in a sense form the elements of a kind of state transition table which drives the statement-scanning algorithm.

III. TRANSFORMATIONS

It was pointed out in passing above that a strictly limited operator-precedence grammar is simply not rich enough to describe these syntactic structures required for MAD/I. There are two immediate demonstrations of this fact, both involving contextual information needed for the resolution of a syntactic type. In the first, a single terminal symbol of V_T is used both to represent a unary operator and to represent a binary operator. The unary plus and minus signs are the most common examples of this, but others can be found in the MAD/I syntax.

Apparently this common syntactical form cannot be described in the obvious fashion in an operator precedence grammar. However, if the two uses of the operator are assigned different names, perhaps the minus sign for the binary case and the %NEG symbol for the unary case, then an operator precedence grammar description is readily apparent. Moreover,

by inspection of the terminal context matrix (see Section 2.4) a simple context-dependent transformation can be synthesized which indicates exactly those contexts in which the minus sign is to be replaced by the metaterminal %NEG. The generalization of this procedure leads to the notion of terminal transformation which will be discussed in detail in following sections.

The second demonstration of the inadequacy of the unenriched operator precedence grammar description for the MAD/I syntax appears at the level of statement parsing. The problem is that, while at the expression level the order of the identification of the various prime phrases parallels the order in which the object code produced will be executed, at the statement level this is not necessarily the case. One might in fact say that the match between the identified syntactic construct and the applicable semantic rules seems to be poor. Another way of saying the same thing is that the basic operator precedence grammar expression scanner is a bottom-up syntax analyzer and such an analyzer works well in a simple algebraic expression environment. On the other hand, the binding structure among the expressional components of a statement can really best be parsed by a goal-oriented top-down analyzer. Techniques for turning the expression scanner inside-out, so to speak, for this purpose will be discussed in following sections. These techniques involve the notion of the precedence transformation, really an extension of the familiar

technique which associates to each instance of an identified prime phrase a macro definition in which the semantic interpretation rules associated with that phrase are expressed.

All context-dependent transformations are identified using the terminal context matrix described in Section 2.4. Properly constructed, the terminal context matrix initiates each type of transformation only under well-defined contextual environments. The hat trick in this procedure, however, is to insure that the excellent error-recovery characteristics inherent in the operator precedence grammar are not unreasonably compromised and that no ambiguities are introduced into the language by virtue of the new syntactic constructions so defined. A specification of the necessary constraints upon the applicable contexts in order that these requirements be satisfied appears elusive using the analysis techniques illustrated herein. On the other hand, a specification of sufficient constraints can be given in certain cases.

3.1 Terminal Transformations

The introduction of context-dependent transformations can be established at two levels: first, consider a sequence of input symbols $a_o a_1 \ldots a_i a_j \ldots (a \in V_T)$ which are input to the compiler. These are extracted in turn from the actual input character stream by the lexical analyzer, so that a_i and a_j for example are identifiers which are represented by descriptors within the compiler. Now, consider the case where the statement scanning algorithm, having just read symbol a_i ,

is about to read symbol a_j. At this point the terminal context matrix is accessed and the integer found at the intersection of the row and column corresponding to a_i and a_j is extracted. The following cases are possible:

1. The integer has the value one, in which case the pair $a_i a_j$ is allowable and the statement scanning algorithm proceeds.

2. The integer has the value zero, in which case the pair $a_i a_j$ represents an error, and a recovery procedure is initiated.

3. The integer has a value other than one or zero and is assumed to identify a built-in transformation which is immediately executed. Such a transformation is called a terminal transformation, and several such are described below.

A terminal transformation is designed to produce a string of terminal symbols in the following manner:

$$ab \rightarrow axb ,$$

where both a and b are terminal symbols and x is an arbitrary string of terminal symbols. (In the useful cases described here x is a single terminal symbol).

In practice, a terminal transformation is constructed by defining an operator precedence grammar with certain additional primitives which cannot by design in the language be elements of an input string. Let the environment of such a

primitive a be represented by xay , where x represents any member of the set of terminal symbols which may occur adjacent to a on the left and y any member of the set which may occur on the right. Now verify that the contexts formed by juxtapositions of an element of x and an element of y are all invalid; that is, these contexts do not occur in the terminal context matrix. When one of these "invalid" contexts is found, then, the introduction of the primitive a in the manner shown is guaranteed to be unambiguous.

In order to preserve the consistency of the language it is necessary to apply the terminal transformation in all equivalent contexts; that is, if both ab and cd are valid terminal strings in the new grammar; then if the terminal transformation is applied in the ab context, it must also be applied in the cd context.

As an example of a practical application of this technique, consider the grammar whose productions are shown in Figure 5. This grammar happens to be used to describe the syntax of the operator and operand fields in an experimental assembler for the PDP-8 and PDP-9 computers. The plus and minus symbols are interpreted as two's complement binary operators and the logical symbols as one's complement bitwise binary operators. The %M symbol stands for the two's complement unary negation operator and the %N symbol stands for the one's complement unary bitwise inversion operator. The %I symbol stands for any operand, either a variable or

```
PRODUCTIONS

001 U1 = %I
002 U1 = U1 %A
003 U2 = U1
004 U2 = %N U2
005 U4 = U2
006 U4 = U4 & U2
007 U5 = U4
008 U5 = U5 | U4
009 U5 = U5 ¬ U4
010 U6 = U5
011 U6 = U7
012 U7 = %N U7
013 U7 = %M U6
014 U6 = U4 & U7
015 U6 = U5 | U7
016 U6 = U5 ¬ U7
017 U8 = U6
018 U8 = U8 * U6
019 U8 = U8 / U6
020 U9 = U8
021 U9 = U9 + U8
022 U9 = U9 - U8
023 U8 = U9
024 U1 = ( U8 )
025 U8 = %L U8 %R

NON-TERMINAL VOCABULARY

U1  U2  U4  U5  U6  U7  U8  U9  U8  U8

TERMINAL VOCABULARY

%I  %A  %N  &  |  ¬  %M  *  /  +  -  (  )  %L  %R
```

Figure 5. Example Grammar—Productions.

a constant. The %L and %R symbols stand for left closure, which marks the bottom of the stack, and right closure, which represents the end-of-statement (card) delimiters respectively. These two symbols are introduced for convenience in error recovery. Finally, the %A stands for an attribute operator used to specify a property of an identifier.

It is the intent in the source language of this experimental assembler to represent both the two's complement binary subtraction operation and the unary negation (%M) operations by the minus sign (-) and both the one's complement bitwise binary subtraction (i.e., exclusive-OR) and unary inversion (%N) operations by the logical-not symbol (¬) . Thus a terminal transformation is to be synthesized which results in the replacement of the - symbol by the %M symbol and the ¬ symbol by the %N symbol in the proper contextual environments.

These environments are readily apparent from the terminal context matrix for this grammar (Figure 6). In this figure note that all the binary operators are in equivalence Class 4 and all the unary operators in equivalence Class 3. Then note that the terminal contexts x%N and x& , where x represents any terminal symbol, %N (a unary operator of Class 3) and & (a binary operator of Class 4) are mutually exclusive. In particular, then, if an "invalid" context y& is found in the source text and furthermore the context y%N is valid, then the terminal transform x&→x%N is indicated.

TERMINAL CONTEXT MATRIX

EQUIVALENCE CLASSES

CL	REP	MEMBERS						
01	%I							
02	%A							
03	%N	%M						
04	&	|	¬	*	/	+	-	
05	(							
06	)							
07	%L							
08	%R							

EQUIVALENCE MATRIX

	%I	%A	%N	&	(	)	%L	%R	
%I		PT		PT		PT		PT	01
%A		PT		PT		PT		PT	02
%N	PT	P	PT	P	PT	P		P	03
&	PT	P	PT	P	PT	P		P	04
(	PT	P	PT	P	PT	P			05
)		PT		PT		PT		PT	06
%L	PT	P	PT	P	PT			P	07
%R									08
	01	02	03	04	05	06	07	08	

Figure 6. Example Grammar—Terminal Context Matrix.

Terminal transformations are implemented within the MAD/I compiler as a macro call, the operands of which include

1. the last terminal symbol scanned a_i ,
2. the terminal symbol next to be read a_j.

The macro may produce the following results:

1. return immediately to the statement-scanning algorithm (a no-operation),
2. replace a_j with a new symbol a_k ,
3. delete a_j , and
4. insert a single symbol x such that x will be the symbol next to be read and a_j the next symbol following x .

The following six terminal transformations are presently implemented within the compiler:

Terminal Error.

The pair $a_i a_j$ is not allowable in the language, nor does it represent a context of any terminal transformation. The macro definition associated with this transformation by convention prints a diagnostic message.

Unary Operation.

The pair $a_i a_j$ represents a context in which a_j would normally be expected to be a unary operator. In this case, however, a_j belongs to the class of binary operators.

The macro definition associated with this transformation by convention:

1. if a_j is the symbol "+" then a_j is deleted,
2. if a_j is the symbol "-" then a_j is replaced by the symbol %NEG representing the unary negation operation.

In other than these two cases a diagnostic message is generated.

Empty Argument.

The pair $a_i a_j$ represents a context in which a_j would normally be expected to be an operand, and furthermore, if x represents such an operand, then the context a_i x a_j is valid in the language. This transformation is involved in several contexts corresponding to missing arguments in function calls and subscription operations The macro definition associated with this transformation by convention inserts a dummy operand between a_i and a_j and this is not considered an error.

Empty Statement.

The pair $a_i a_j$ represents a context in which a_j is normally expected to be a statement, and furthermore, if x represents such a statement, then the context a_i x a_j is valid in the language. The macro definition associated with this transformation by convention inserts a dummy operand between a_i and a_j and this is not considered an error.

Empty Declarative List Element.

The pair $a_i a_j$ represents a context in which a_j is normally expected to be a declarative list element (see Section 4.2), and furthermore, if x represents such an element, then the context a_i x a_j is valid in the language. This transformation is used during the scan of those declarations which apply default attributes to the program. The macro definition associated with this transformation by convention inserts the %DEFAULT operand between a_i and a_j and, if a_j is the symbol ";" , this is not considered an error.

Empty Executable List Element.

The pair $a_i a_j$ represents a context in which a_j is normally expected to be an executable list element (see Section 4.2), and furthermore, if x represents such an element, then the context a_i x a_j is valid in the language. The macro definition associated with this transformation by convention inserts a dummy operand between a_i and a_j and this is not considered an error.

The %TAG Transformation.

Although classed as a terminal transformation, the %TAG transformation exhibits a special behavior. The pair $a_i a_j$ represents one of the contexts ")(" or "%ID(". The %TAG transformation causes a metaterminal symbol x to be inserted between a_i and a_j such that the context a_i x a_j is valid in the kernel grammar. There are two interpretations

of this transformation depending upon its occurrence in a declarative list element or an executable list element. If the %TAG transformation occurs in a declarative list element, then an implicit attribute assignment is indicated which interprets the list elements within the parentheses on the right as an attribute structure to be attached to the operand (possibly a list enclosed in parentheses) on the left The nature of this interpretation can depend both on the name of the declaration statement in which this occurrence is embedded and on the name of the macro definition invoked by the transformation. In this case, the name is given as an argument to the statement-scanning algorithm.

If the %TAG transformation occurs in an executable list element, then an implicit subscription operation is indicated which interprets the list elements within the parentheses on the right as an argument to a component selection function which identifies a particular component of an array during execution. In this case also, the macro name invoked by the transformation is given as an argument to the statement-scanning algorithm.

The above transformations provide some enrichment of the kernel grammar without materially affecting its generative power. Note that although the contextual environments which cause these transformations to be invoked are not normally definable during compilation, the macro definitions associated with the names mentioned are of course definable Thus the behavior effected in the individual cases may be altered by definitional procedures.

3.2 Precedence Transformations

Although the terminal transformations described in the preceding section provide some additional power to the basic expression-scanning algorithm, the power is principally concentrated in reducing the nuisance value of the language by allowing some syntactic "cheating" in the specification of the language. On the other hand, the basic analytical problem inherent in a bottom-up parsing algorithm remains: it is exceedingly difficult to specify the syntax of a complicated statement involving several constituent expressions without doing much violence to its semantic interpretation rules.

The approach taken in the design of the MAD/1 compiler has been to represent certain syntactic forms which have been parsed by the expression-scanning algorithm as an instance of a metaterminal symbol which is an element of the kernel grammar. This technique involves the identification by means of a terminal transformation of the initial character or prefix of that structure which, when parsed, will become the metaterminal symbol. Once such a context has been identified, the basic scanning process recurses in such a way as to exhibit a top-down behavior. In other words, the identification of the metaterminal becomes a process directed by commands embedded within a macro definition, and this process can be obviously context-dependent. Some of the macro commands can cause the basic scanning process to resume its precedence-directed scan at this lower level, but with the additional

requirement that a goal-directed behavior be realized. When the syntactic structure representing the metaterminal symbol is completely parsed, perhaps requiring several goal-directed scanner calls, a nonterminal symbol representing the metaterminal symbol is generated and the scanner pops up to the original statement scan level

The manner in which the goal-directed syntactic scan is realized using a precedence-directed scanning algorithm is obviously the key to the success of this technique. This is done candidly, by a seat-of-the-pants combination of rule-bending and judicious use of what are called here precedence transformations.

The explanation of how this is done requires some superficial explanation of the manner in which the statement-scanning algorithm operates. The algorithm, patterned after those suggested by Bauer and Samelson, Arden and Galler, Floyd, and several others, makes use of a compile-time stack in which symbols are stored during the parsing process This stack at each instance during the scan contains a sequence of symbols, each symbol representing either an operator or a non-terminal symbol. At the top of the stack is a nonterminal symbol X (possibly null), and immediately below this is at least one terminal symbol a_k not of the operand class. Let this terminal symbol be identified by a_k Then consider the sequence of symbols $a_o a_1 \ldots a_k \ldots a_j$ which are input to the translator. Now, having just read a_j, the statement-

scanning algorithm establishes a precedence relation between a_k on one hand and a_j on the other. Note that the symbols between a_k and a_j already have been read and the terminal pairs established as allowable. Thus all terminal transformations have been completed at this point. Now, when a_k and a_j are compared in the precedence relation, the second layer of the terminal context matrix is accessed and the integer found at the intersection of the row and column corresponding to a_k and a_j is extracted. The following cases are possible:

1. The integer has the value one, in which case the pair $a_k a_j$ is contained in a nonvoid precedence relation and the statement scanning algorithm proceeds.

2. The integer has the value zero, in which case the pair $a_k a_j$ represents an error and a recovery procedure is initiated.

3. The integer has a value other than one or zero and is assumed to identify a macro transformation which is immediately executed. Such a transformation is called a precedence transform, and several such are described below.

A precedence transformation is implemented within the MAD/I compiler by a macro definition in the following manner: Let a_k and a_j represent the terminal symbols compared in the precedence relation in the manner described above. Let the precedence context $a_k a_j$ be selected as an environment for a precedence transformation, and furthermore

require that $a_k \lessdot a_j$. Then the precedence transformation associated with the name a_j will:

1. stack the representative of the equivalence class containing a_j (or a representative of another equivalent class which obeys the same precedence and terminal relations in the "left context" of a_j);

2. initiate the statement-scanning algorithm at the next lower level to scan the arguments of the statement identified by a_j ; and, finally,

3. replace X and a_j on the stack with a nonterminal symbol which represents the result of the transformation.

The integrity of the kernel language is not compromised if at least the following conditions are satisfied: Let T be a metaterminal symbol such that

1. in all allowable contexts T_1U_1T , the pair T_1T is selected as an environment for the same transformation and, in all of these contexts, $T_1 \lessdot T$;

2. in all allowable contexts TU_2T_2 , $T \doteq T_2$

Thus the macro associated with the transformation bears the responsibility of "positioning" the input text pointer properly before surrendering to the higher statement scanning level at which it was invoked. Convenient rules for accomplishing this involve the tables of left and right terminal delimiters developed in Section 2.4

Four precedence transformations are recognized within the compiler. Three out of these four are essentially fixed within the compiler and are not subject to redefinition. The fourth is implemented as a macro call and is a good example of the statement definition capability of the compiler. All of these will be discussed briefly below.

Parenthesized List Element

Note the productions containing the parenthesized list element (PLS) in Figure 7, Section 4.1, as a left part. All of these productions take the form (X) where X is a nonterminal symbol. Furthermore, the only occurrence of parentheses are in these productions. The parenthesized list element transformation in fact performs the operation (X)→PLS. This transformation could have been performed as a macro operation and is performed as a compiler operation only for the sake of convenience.

List Element.

All argument lists in function calls and subscription operations are presumed to be linear; that is, no tree-like structures are allowed. The commas which separate the list items are then superfluous. The list transformation performs the operation

' X , → X (

where X is a nonterminal symbol. Both the parenthesized

list transformation and the list transformation are expected to evolve as richer structures are incorporated into the compiler.

Statement Keyword.

Several terminal symbol syntactic classes are designated as statement keyword classes. Among these are the symbols of the %SIMP, %COMP, %DECL, %LIST, %ATRB, @ and: classes (see Figure 7, Section 4.1) The first four of these represent symbols most likely to designate an identifying keyword of a statement. Inspection of the precedence matrix for the kernel grammar (Figure 13, Section 4.1) reveals that for every symbol a which can occur in a precedence relation on the left along with a statement keyword b on the right, that

$$a \lessdot b \ .$$

Each instance of this type is chosen as an instance of a keyword transformation, which causes a macro definition to be invoked, the name of which is the keyword itself. The macro definition generates connectives as required and calls upon the statement-scanning algorithm at a lower level to scan the arguments of the prefix and scope Each time the statement-scanning algorithm is called, an element of one of the statement keyword classes is stacked, depending upon the nonterminal symbol expected as the argument, and according to the

following table, which is a subset of the terminal delimiter tables (see Figures 13 and 19, Section 4.1)

Keyword Class	Nonterminal	List Separator	Statement Separator
%SIMP	STM	none	) ; %END %RC
%COMP	STM	;	%END
%LIST	LST	,	) ; %END %RC
%DECL	LSD	,	) ; %END %RC

The statement-scanning algorithm parses the succeeding text until an ending condition is recognized.

End.

This transformation complements the above keyword transformation by providing a mechanism for returning the statement-scanning algorithm to the macro which initiated it. The ending condition is recognized when a precedence comparison is made between the keyword stacked upon initiation of the scan (see above) and one of the symbols in either the list separator or statement separator columns of the above table. The elements in the list separator and statement separator columns of this table are determined as follows: if a is a keyword, and b is a right terminal delimiter of U in a production

$$U_1 \rightarrow aUb \ ,$$

then b is a list separator if $a \doteq b$, and b is a statement separator if $b \gtrdot a$. The statement-scanning algorithm

will then return control to the macro which initiated its operation. The macro now has the option of continuing the scan by again calling the statement-scanning algorithm or returning to the statement-scanning algorithm at the next higher level, depending upon whether the terminating symbol belongs to the list separator or statement separator classes.

IV. A KERNEL GRAMMAR FOR MAD/I

In establishing a production system for MAD/I, several considerations are apparent. First, of course, the language generated must be unambiguous. Second, the productions must provide some "handles" so that context-dependent transformations can be strategically applied Finally, the productions must bear a relationship to those program constructions most familiar in MAD, that is the expression, the statement, and the program.

The first requirement is satisfied by insisting that the kernel production system represent an operator precedence grammar. Certain context-dependent transformations can be applied to the source language to preserve the integrity of the language in each exceptional instance. Some of these transformations generate metaterminal symbols which by design cannot occur in the input text. These provide the "handles" satisfying the second condition. The third condition is satisfied rather naturally by requiring that the productions take on the forms of Rules 1-6 (see Section 2.2).

4.1 An Operator Precedence Kernel Grammar

Figure 7 shows a table of productions which constitute an operator precedence kernel grammar of MAD/I. These productions are divided roughly into five groups:

1. the program primitives,
2. the assignment statement,
3. the list statement,
4. the declaration statement, and
5. the program structure

The collection of all those symbols on the left of the equal sign = , which corresponds to the more familiar right arrow → , corresponds to the nonterminal vocabulary V_N . The complement of V_N relative to all the symbols occurring either on the left or the right of the equal sign is the terminal vocabulary V_T . These two sets are enumerated in Figure 8. Note that in these and other tables of this Section only the first three characters of each symbol are shown.

Only those nonterminal symbols which do not begin with an X are significant in the discussion; these are interpreted roughly as follows (see also Section 4.2):

IDR - Stands for either an identifier extracted by the lexical scan or a parenthesized list.

```
PRODUCTIONS

PROGRAM PRIMITIVES

001 XL = %IDN
002 XL = %LP PLS
003 IDR = XL
004 IDR = IDR @ XL
005 XM = IDR
006 XM = XM %TAG IDR
007 XM = XM %KEY
008 DES = XM
009 DES = DES . XM

ASSIGNMENT STATEMENT

010 X1 = DES
011 X1 = .ABS. X1
012 X2 = X1
013 X2 = X2 .LS. X1
014 X2 = X2 .RS. X1
015 X3 = X2
016 X3 = X3U
017 X3U = .ABS. X3U
018 X3 = X2 .LS. X3U
019 X3 = X2 .RS. X3U
020 X3U = .N. X3
021 X4 = X3
022 X4 = X4 .A. X3
023 X5 = X4
024 X5 = X5 .V. X4
025 X5 = X5 .EV. X4
026 X6 = X5
027 X6 = X6 ** X5
028 X7 = X6
029 X7 = X7U
030 X7U = .ABS. X7U
031 X7 = X2 .LS. X7U
032 X7 = X2 .RS. X7U
033 X7U = .N. X7U
034 X7 = X4 .A. X7U
035 X7 = X5 .V. X7U
036 X7 = X5 .EV. X7U
037 X7 = X6 ** X7U
038 X7U = %NEG X7
039 X8 = X7
040 X8 = X8 * X7
041 X8 = X8 / X7
042 X9 = X8
043 X9 = X9 + X8
044 X9 = X9 - X8
045 XA = X9
046 XA = XA = X9
```

Figure 7. Kernel Productions for MAD/I.

[Page 1 of 3]

```
047 XA = XA ¬= X9
048 XA = XA > X9
049 XA = XA >= X9
050 XA = XA < X9
051 XA = XA <= X9
052 XB = XA
053 XB = XBU
054 XBU = .ABS. XBU
055 XB = X2 .LS. XBU
056 XB = X2 .RS. XBU
057 XBU = .N. XBU
058 XB = X4 .A. XBU
059 XB = X5 .V. XBU
060 XB = X5 .EV. XBU
061 XB = X6 ** XBU
062 XBU = %NEG XBU
063 XB = X8 * XBU
064 XB = XB / XBU
065 XB = X9 + XBU
066 XB = X9 - XBU
067 XB = XA = XBU
068 XB = XA ¬= XBU
069 XB = XA > XBU
070 XB = XA >= XBU
071 XB = XA < XBU
072 XB = XA <= XBU
073 XBU = ¬ XB
074 XC = XB
075 XC = XC & XB
076 XD = XC
077 XD = XD | XC
078 XD = XD .FXOR. XC
079 XE = XD
080 XE = XE .THEN. XD
081 XF = XE
082 XF = XF .FQV. XE
083 ASN = XF
084 ASN = DES == ASN
085 STM = ASN

LIST STATEMENT

086 XH = DES
087 XH = XH ,,, DES
088 XJ = XH
089 XJ = ASN
090 LST = XJ
091 LST = LST , XJ
092 STM = %LIST LST
```

Figure 7. Kernel Productions for MAD/I.
[Page 2 of 3]

```
DECLARATION STATEMENT

093 XK = IDR
094 XK = XK %ATRB IDR
095 LSD = XK
096 LSD = LSD , XK
097 STM = %DECL LSD

PROGRAM STRUCTURE

098 PLS = ( LSD )
099 PLS = ( LST )
100 PLS = ( STM )
101 STM = DES : STM
102 STM = %SIMP STM
103 STL = STM
104 STL = STL ; STM
105 STM = %COMP STL %END
106 PGM = %LC STL %RC
```

Figure 7. Kernel Productions for MAD/I.
[Page 3 of 3]

NONTERMINAL VOCABULARY

```
XL   IDR XM   DES X1   X2   X3   X3U ^4   X5   X6   X7   X7U X8   X9   XA   XB
XBU  XC  XD   XE  XF   ASN  STM  XH  ^J   LST  XK   LSD  PLS  STL  PGM
```

TERMINAL VOCABULARY

```
%ID  %LP  @   %TA  %KE  .    .AB  .LS  .RS  .N.  .A.  .V.  .EV  **   %NE  *    /
+    -    =   ¬=   >    >=   <    <=   ¬    &    |    .EX  .TH  .EQ  ==   ...  ,
%LI  %AT  %DE (    )    :    %SI  ;    %CO  %EN  %LC  %RC
```

Figure 8. Kernel Vocabulary.

DES - Stands for a designator, that is an identifier, with or without attribute notation, possibly subscripted, or the result of a function evaluation.

ASN - Stands for an assignment, that is an expression containing the usual arithmetic and logical operators and in addition the substitution operator == .

LST - Stands for a list element, that is a list of elements each of which is either an assignment or an instance of block notation.

LSD - Stands for a declarative list element, that is a list of elements each of which is either an identifier or a special notation used in the declaration statements.

PLS - Stands for a parenthesized list, that is a list of list elements, declarative list elements, or statements.

STM - Stands for a statement, that is either an assignment or a list preceded by a keyword.

STL - Stands for a statement list, that is a list of statements separated by a semicolon ; .

PGM - Stands for a program, that is a statement list delimited by the metaterminal symbols for left and right closure.

The various terminal symbols are interpreted as in Figure 9. Note that all those symbols preceded by the percent sign % are identified as metaterminal symbols within the compiler and are created as the result of context-dependent transformations. Note further that some communications equipment cannot produce or recognize some of the special characters used here In these cases, synonyms constructed of names surrounded by periods are provided

The remaining figures in this section correspond to those tables and matrices developed for Floyd's simple grammar in Sections 2.3 and 2.4. Figures 10 and 11 show respectively the table of left and right terminal derivatives, and Figure 12 shows the equivalence classes assigned to the precedence matrix and the members of each class Figure 13 shows the precedence matrix itself. Note that only the first two characters of the symbol representing each class are shown

Figure 14 shows the equivalence classes assigned to the terminal matrix, and Figure 15 shows the terminal matrix itself. Figure 16 shows the equivalence classes assigned the terminal context matrix, and Figure 17 shows the terminal context matrix itself. Figures 18 and 19 show the table of left and right terminal delimiters respectively. Finally, Figure 9 summarizes the equivalence class assignments for all of the matrices and in addition shows the left and right precedence functions assigned to each terminal symbol.

DESCRIPTOR ASSIGNMENTS

TRM SYM	RULE FORM	PRC MTRX	TRM MTRX	CTX MTRX	REL F	REL G	
%ID	1	1	1	1	33	34	operand
%LP	1	2	2	2	33	34	literal prefix operator
@	6	3	3	3	33	32	attribute notation operator
%TAG	6	4	4	4	31	30	tag operator
.	6	6	4	4	29	28	function operator
%KEY	3	5	5	5	31	30	component selection operator
.ABS.	2	7	6	6	27	28	absolute value operator
.N.	2	9	6	6	25	28	bitwise logical NOT operator
%NEG	2	13	6	6	19	28	negation operator
¬	2	17	6	6	13	28	logical NOT operator
.LS.	6	8	7	7	27	26	bitwise left shift operator
.RS.	6	8	7	7	27	26	bitwise right shift operator
.A.	6	10	7	7	25	24	bitwise logical AND operator
.V.	6	11	7	7	23	22	bitwise logical OR operator
.EV.	6	11	7	7	23	22	bitwise logical EXCLUSIVE OR operator
**	6	12	7	7	21	20	exponentiation operator
*	6	14	7	7	19	18	multiplication operator
/	6	14	7	7	19	18	division operator
+	6	15	7	7	17	16	addition
-	6	15	7	7	17	16	subtraction
=	6	16	7	7	15	14	EQUAL relational operator
¬=	6	16	7	7	15	14	INEQUAL relational operator
>	6	16	7	7	15	14	GREATER THAN relational operator
>=	6	16	7	7	15	14	GREATER THAN OR EQUAL relational operator
<	6	16	7	7	15	14	LESS THAN relational operator
<=	6	16	7	7	15	14	LESS THAN OR EQUAL relational operator
&	6	18	7	7	13	12	logical AND operator
|	6	19	7	7	11	10	logical OR operator
.EXOR.	6	19	7	7	11	10	logical EXCLUSIVE OR operator
.THEN.	6	20	7	7	9	8	logical IMPLICATION operator
.EQV.	6	21	7	7	7	6	logical EQUIVALENCE operator
= =	6	22	7	8	5	6	substitution operator
. . .	6	23	4	9	7	6	block notation operator
,	6	24	7	10	5	4	list delimiter
%LIST	2	25	8	11	3	4	list statement
%ATRB	6	26	3	12	7	6	attribute expression
%DECL	2	27	9	13	3	4	declaration statement
(	4	28	10	14	1	34	left paren
)	4	29	11	15	33	1	right paren
:	6	30	12	16	3	4	label delimiter
%SIMP	2	31	13	17	3	4	simple statement
;	6	32	14	18	3	2	statement delimiter
%COMP	4	33	13	19	1	4	compound statement
%END	4	34	15	20	3	1	END delimiter
%LC	4	35	16	21	1	1	left closure
%RC	4	36	17	22	1	1	right closure

Figure 9. Descriptor Assignment.

LEFT TERMINAL DERIVATIVES

```
NTC TERMINAL CHARACTERS
XL  %ID %LP
IDR %ID %LP @
XM  %ID %LP @   %TA %KE
DE: %ID %LP @   %TA %KE .
X1  %ID %LP @   %TA %KE .   .AB
X2  %ID %LP @   %TA %KE .   .AB .LS .RS
X3  %ID %LP @   %TA %KE .   .AB .LS .RS .N.
X3U .AB .N.
X4  %ID %LP @   %TA %KE .   .AB .LS .RS .N. .A.
X5  %ID %LP @   %TA %KE .   .AB .LS .RS .N. .A. .V. .EV
X6  %ID %LP @   %TA %KE .   .AB .LS .RS .N. .A. .V. .EV **
X7  %ID %LP @   %TA %KE .   .AB .LS .RS .N. .A. .V. .EV **  %NE
X7U .AB .N. %NE
X8  %ID %LP @   %TA %KE .   .AB .LS .RS .N. .A. .V. .EV **  %NE *
    /
X9  %ID %LP @   %TA %KE .   .AB .LS .RS .N. .A. .V. .EV **  %NE *
    /   +   -
XA  %ID %LP @   %TA %KE .   .AB .LS .RS .N. .A. .V. .EV **  %NE *
    /   +   -   =   ¬=  >   >=  <   <=
XB  %ID %LP @   %TA %KE .   .AB .LS .RS .N. .A. .V. .EV **  %NE *
    /   +   -   =   ¬=  >   >=  <   <=  ¬
XBU .AB .N. %NE ¬
XC  %ID %LP @   %TA %KE .   .AB .LS .RS .N. .A. .V. .EV **  %NE *
    /   +   -   =   ¬=  >   >=  <   <=  ¬   &
XD  %ID %LP @   %TA %KE .   .AB .LS .RS .N. .A. .V. .EV **  %NE *
    /   +   -   =   ¬=  >   >=  <   <=  ¬   &   |   .EX
XE  %ID %LP @   %TA %KE .   .AB .LS .RS .N. .A. .V. .EV **  %NE *
    /   +   -   =   ¬=  >   >=  <   <=  ¬   &   |   .EX .TH
XF  %ID %LP @   %TA %KE .   .AB .LS .RS .N. .A. .V. .EV **  %NE *
    /   +   -   =   ¬=  >   >=  <   <=  ¬   &   |   .EX .TH .EQ
ASN %ID %LP @   %TA %KE .   .AB .LS .RS .N. .A. .V. .EV **  %NE *
    /   +   -   =   ¬=  >   >=  <   <=  ¬   &   |   .EX .TH .EQ ==
STM %ID %LP @   %TA %KE .   .AB .LS .RS .N. .A. .V. .EV **  %NE *
    /   +   -   =   ¬=  >   >=  <   <=  ¬   &   |   .EX .TH .EQ ==
    %LI %DE :   %SI %CO
XH  %ID %LP @   %TA %KE .   ...
XJ  %ID %LP @   %TA %KE .   .AB .LS .RS .N. .A. .V. .EV **  %NE *
    /   +   -   =   ¬=  >   >=  <   <=  ¬   &   |   .EX .TH .FQ ==
    ...
LST %ID %LP @   %TA %KE .   .AB .LS .RS .N. .A. .V. .EV **  %NE *
    /   +   -   =   ¬=  >   >=  <   <=  ¬   &   |   .EX .TH .EQ ==
    ... ,
XK  %ID %LP @   %AT
LSD %ID %LP @   ,   %AT
PLS (
STL %ID %LP @   %TA %KE .   .AB .LS .RS .N. .A. .V. .EV **  %NE *
    /   +   -   =   ¬=  >   >=  <   <=  ¬   &   |   .EX .TH .EQ ==
    %LI %DE :   %SI :   %CO
PGM %LC
```

Figure 10. Left Terminal Derivatives.

RIGHT TERMINAL DERIVATIVES

```
NTC TERMINAL CHARACTERS
XL   %ID %LP )
IDR  %ID %LP @   )
XM   %ID %LP @   %TA %KE )
DES  %ID %LP @   %TA %KE .   )
X1   %ID %LP @   %TA %KE .   .AB )
X2   %ID %LP @   %TA %KE .   .AB .LS .RS )
X3   %ID %LP @   %TA %KE .   .AB .LS .RS .N. )
X3U  %ID %LP @   %TA %KE .   .AB .LS .RS .N. )
X4   %ID %LP @   %TA %KE .   .AB .LS .RS .N. .A. )
X5   %ID %LP @   %TA %KE .   .AB .LS .RS .N. .A. .V. .EV )
X6   %ID %LP @   %TA %KE .   .AB .LS .RS .N. .A. .V. .EV **  )
X7   %ID %LP @   %TA %KE .   .AB .LS .RS .N. .A. .V. .EV **  %NE )

X7U  %ID %LP @   %TA %KE .   .AB .LS .RS .N. .A. .V. .EV **  %NE )

X8   %ID %LP @   %TA %KE .   .AB .LS .RS .N. .A. .V. .EV **  %NE *
     /   )
X9   %ID %LP @   %TA %KE .   .AB .LS .RS .N. .A. .V. .EV **  %NE *
     /   +   -   )
XA   %ID %LP @   %TA %KE .   .AB .LS .RS .N. .A. .V. .EV **  %NE *
     /   +   -   =   ¬=  >   >=  <   <=  )
XB   %ID %LP @   %TA %KE .   .AB .LS .RS .N. .A. .V. .EV **  %NE *
     /   +   -   =   ¬=  >   >=  <   <=  ¬   )
XBU  %ID %LP @   %TA %KE .   .AB .LS .RS .N. .A. .V. .EV **  %NE *
     /   +   -   =   ¬=  >   >=  <   <=  ¬   )
XC   %ID %LP @   %TA %KE .   .AB .LS .RS .N. .A. .V. .EV **  %NE *
     /   +   -   =   ¬=  >   >=  <   <=  ¬   &   )
XD   %ID %LP @   %TA %KE .   .AB .LS .RS .N. .A. .V. .EV **  %NE *
     /   +   -   =   ¬=  >   >=  <   <=  ¬   &   |   .EX )
XE   %ID %LP @   %TA %KE .   .AB .LS .RS .N.  A. .V. .EV **  %NE *
     /   +   -   =   ¬=  >   >=  <   <=  ¬   &   |   .EX .TH )
XF   %ID %LP @   %TA %KE .   .AB .LS .RS .N. .A. .V. .EV **  %NE *
     /   +   -   =   ¬=  >   >=  <   <=  ¬   &   |   .EX .TH .EQ )
ASN  %ID %LP @   %TA %KE .   .AB .LS .RS .N. .A. .V. .EV **  %NE *
     /   +   -   =   ¬=  >   >=  <   <=  ¬   &   |   .EX .TH .EQ ==
     )
STM  %ID %LP @   %TA %KE .   .AB .LS .RS .N. .A. .V. .EV **  %NE *
     /   +   -   =   ¬=  >   >=  <   <=  ¬   &   |   .EX .TH .EQ ==
     ... ,   %LI %AT %DE )   :   %SI %EN
XH   %ID %LP @   %TA %KE .   ... )
XJ   %ID %LP @   %TA %KE .   .AB .LS .RS .N. .A. .V. .EV **  %NE *
     /   +   -   =   ¬=  >   >=  <   <=  ¬   &   |   .EX .TH .EQ ==
     .. )
LST  %ID %LP @   %TA %KE .   .AB .LS .RS .N. .A. .V. .EV **  %NE *
     /   +   -   =   ¬=  >   >=  <   <=  ¬   &   |   .EX .TH .EQ ==
     ... ,   )
XK   %ID %LP @   %AT )
LSD  %ID %LP @   ,   %AT )
PLS  ;
STL  %ID %LP @   %TA %KE .   .AB .LS .RS .N. .A. .V. .EV **  %NE *
     /   +   -   =   ¬=  >   >=  <   <=  ¬   &   |   .EX .TH .EQ ==
     ... ,   %LI %AT %DE )   :   %SI ;   %EN
PGM  %RC
```

Figure 11. Right Terminal Derivatives.

PRECEDENCE MATRIX

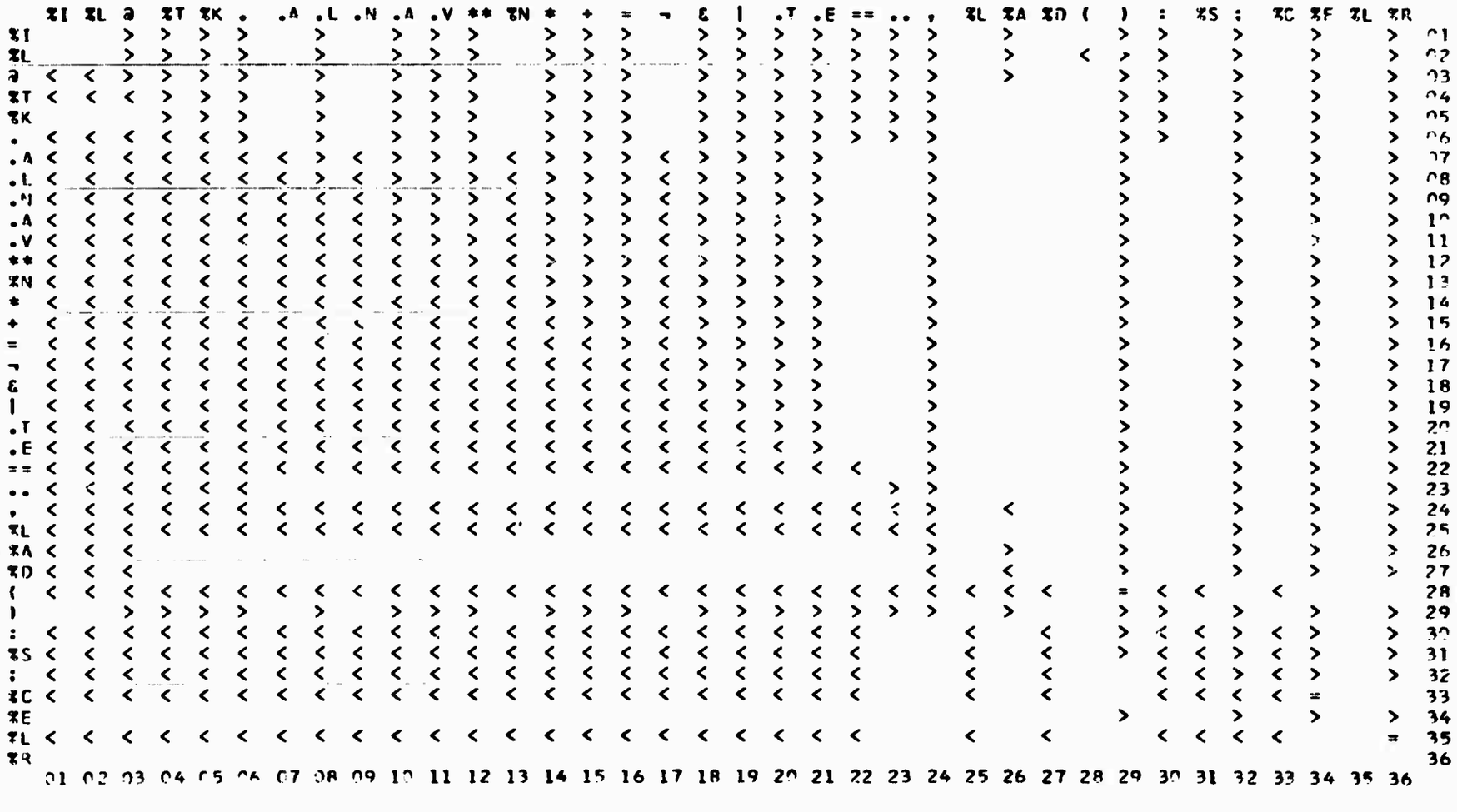
EQUIVALENCE CLASSES

CL	REP	MEMBERS
01	%ID	
02	%LP	
03	@	
04	%TA	
05	%KE	
06	.	
07	.AB	
08	.LS	.RS
09	.N.	
10	.A.	
11	.V.	.EV
12	**	
13	%NE	
14	*	/
15	+	-
16	=	¬= > >= < <=
17	¬	
18	&	
19	|	.EX
20	.TH	
21	.EQ	
22	==	
23	...	
24	,	
25	%LI	
26	%AT	
27	%DE	
28	(	
29	)	
30	:	
31	%SI	
32	;	
33	%CO	
34	%EN	
35	%LC	
36	%RC	

Figure 12. Precedence Equivalence Classes.

PRECEDENCE MATRIX

EQUIVALENCE MATRIX



	%I	%L	@	%T	%K	.	.A	.L	.N	.A	.V	**	%N	*	+	=	¬	&	|	.T	.E	==	..	,	%L	%A	%D	(	)	:	%S	;	%C	%F	%L	%R	
%I			>	>	>	>		>		>	>	>		>	>	>		>	>	>	>	>	>	>		>			>	>		>		>		>	01
%L			>	>	>	>		>		>	>	>		>	>	>		>	>	>	>	>	>	>		>		<	>	>		>		>		>	02
@	<	<	>	>	>	>		>		>	>	>		>	>	>		>	>	>	>	>	>	>		>			>	>		>		>		>	03
%T	<	<	<	>	>	>		>		>	>	>		>	>	>		>	>	>	>	>	>	>		>			>	>		>		>		>	04
%K				>	>	>		>		>	>	>		>	>	>		>	>	>	>	>	>	>		>			>	>		>		>		>	05
.	<	<	<	<	<	>		>		>	>	>		>	>	>		>	>	>	>	>	>	>		>			>	>		>		>		>	06
.A	<	<	<	<	<	<	<	>	<	>	>	>	<	>	>	>	<	>	>	>	>			>					>			>		>		>	07
.L	<	<	<	<	<	<	<	>	<	>	>	>	<	>	>	>	<	>	>	>	>			>					>			>		>		>	08
.N	<	<	<	<	<	<	<	<	<	>	>	>	<	>	>	>	<	>	>	>	>			>					>			>		>		>	09
.A	<	<	<	<	<	<	<	<	<	>	>	>	<	>	>	>	<	>	>	>	>			>					>			>		>		>	10
.V	<	<	<	<	<	<	<	<	<	<	>	>	<	>	>	>	<	>	>	>	>			>					>			>		>		>	11
**	<	<	<	<	<	<	<	<	<	<	<	>	<	>	>	>	<	>	>	>	>			>					>			>		>		>	12
%N	<	<	<	<	<	<	<	<	<	<	<	<	<	>	>	>	<	>	>	>	>			>					>			>		>		>	13
*	<	<	<	<	<	<	<	<	<	<	<	<	<	>	>	>	<	>	>	>	>			>					>			>		>		>	14
+	<	<	<	<	<	<	<	<	<	<	<	<	<	<	>	>	<	>	>	>	>			>					>			>		>		>	15
=	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	>	<	>	>	>	>			>					>			>		>		>	16
¬	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	>	>	>	>			>					>			>		>		>	17
&	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	>	>	>			>					>			>		>		>	18
|	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	>	>			>					>			>		>		>	19
.T	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	>			>					>			>		>		>	20
.E	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	>			>					>			>		>		>	21
==	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<		>					>			>		>		>	22
..	<	<	<	<	<	<																	>	>					>			>		>		>	23
,	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	>	>		<			>			>		>		>	24
%L	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<					>			>		>		>	25
%A	<	<	<																					>		>			>			>		>		>	26
%D	<	<	<																					<		<			>			>		>		>	27
(	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<		=	<	<		<				28
)			>	>	>	>		>		>	>	>		>	>	>		>	>	>	>	>	>	>		>			>	>		>		>		>	29
:	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<			<		<		>	<	<	>	<	>		>	30
%S	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<			<		<		>	<	<	>	<	>		>	31
;	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<			<		<			<	<	>	<	>		>	32
%C	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<			<		<			<	<	<	<	=			33
%E																													>			>		>		34
%L	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<			<		<			<	<	<	<			=	35
%R																																				36
	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	

Figure 13. Precedence Matrix.

TERMINAL PAIR MATRIX

EQUIVALENCE CLASSES

CL	REP	MEMBERS														
01	%ID															
02	%LP															
03	@	%AT														
04	%TA	.	...													
05	%KE															
06	.AB	.N.	%NE	¬												
07	.LS	.RS	.A.	.V.	.EV	**	*	/	+	-	=	¬=	>	>=	<	<=
	&	|		.EX	.TH	.EQ	==	,							
08	%LI															
09	%DE															
10	(															
11	)															
12	:															
13	%SI	%CO														
14	;															
15	%EN															
16	%LC															
17	%RC															

Figure 14. Terminal Pair Equivalence Classes.

TERMINAL PAIR MATRIX

EQUIVALENCE MATRIX

	%I	%L	@	%T	%K	.A	.L	%L	%D	(	)	:	%S	;	%E	%L	%R	
%I			T	T	T		T				T	T		T	T		T	01
%L										T								02
@	T	T																03
%T	T	T																04
%K				T	T		T				T	T		T	T		T	05
.A	T	T				T												06
.L	T	T				T												07
%L	T	T				T												08
%D	T	T																09
(	T	T				T		T	T				T					10
)			T	T	T		T				T	T		T	T		T	11
:	T	T				T		T	T				T					12
%S	T	T				T		T	T				T					13
;	T	T				T		T	T				T					14
%E											T			T	T		T	15
%L	T	T				T		T	T				T					16
%R																		17
	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	

Figure 15. Terminal Pair Matrix.

TERMINAL CONTEXT MATRIX

EQUIVALENCE CLASSES

CL	REP	MEMBERS
01	%ID	
02	%LP	
03	@	
04	%TA	.
05	%KE	
06	.AB	.N. %NE ¬
07	.LS	.RS .A. .V. .EV ** * / + - = ¬= > >= < <= & | .EX .TH .EQ
08	==	
09	...	
10	,	
11	%LI	
12	%AT	
13	%DE	
14	(	
15	)	
16	:	
17	%SI	
18	;	
19	%CO	
20	%EN	
21	%LC	
22	%RC	

Figure 16. Terminal Context Equivalence Classes.

TERMINAL CONTEXT MATRIX

EQUIVALENCE MATRIX

	%I	%L	@	%T	%K	.A	.L	==	..	,	%L	%A	%D	(	)	:	%S	;	%C	%E	%L	%R	
%I			PT	PT	PT		PT	PT	PT	PT		PT			PT	PT		PT		PT		PT	01
%L			P	P	P		P	P	P	P		P		PT	P	P		P		P		P	02
@	PT	PT	P	P	P		P	P	P	P		P			P	P		P		P		P	03
%T	PT	PT	P	P	P		P	P	P	P					P	P		P		P		P	04
%K			PT	PT			PT	PT	PT	PT					PT	PT		PT		PT		PT	05
.A	PT	PT	P	P	P	PT	P			P					P			P		P		P	06
.L	PT	PT	P	P	P	PT	P			P					P			P		P		P	07
==	PT	PT	P	P	P	PT	P	P		P					P			P		P		P	08
..	PT	PT	P	P	P				P	P					P			P		P		P	09
,	PT	PT	P	P	P	PT	P	P	P	P		P			P			P		P		P	10
%L	PT	PT	P	P	P	PT	P	P	P	P					P			P		P		P	11
%A	PT	PT	P							P		P			P			P		P		P	12
%D	PT	PT	P							P		P			P			P		P		P	13
(	PT	PT	P	P	P	PT	P	P	P	P	PT	P	PT		P	P	PT		PT				14
)			PT	PT	PT		PT	PT	PT	PT		PT			PT	PT		PT		PT		PT	15
:	PT	PT	P	P	P	PT	P	P			PT		PT		P	P	PT	P	PT	P		P	16
%S	PT	PT	P	P	P	PT	P	P			PT		PT		P	P	PT	P	PT	P		P	17
;	PT	PT	P	P	P	PT	P	P			PT		PT			P	PT	P	PT	P		P	18
%C	PT	PT	P	P	P	PT	P	P			PT		PT			P	PT	P	PT	P			19
%E															PT			PT		PT		PT	20
%L	PT	PT	P	P	P	PT	P	P			PT		PT			P	PT	P	PT			P	21
%R																							22
	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	

Figure 17. Terminal Context Matrix.

RIGHT TERMINAL DELIMITERS

NTC	TERMINAL CHARACTERS															
XL	@	%TA	%KE	.	.LS	.RS	.A	.V.	.EV	**	*	/	+	-	=	¬=
	>	>=	<	<=	&	|	.EX	.TH	.EQ	==	...	,	%AT	)	:	;
	%EN	%RC														
IOR	@	%TA	%KE	.	.LS	.RS	.A.	.V.	.EV	**	*	/	+	-	=	¬=
	>	>=	<	<=	&	|	.EX	.TH	.EQ	==	...	,	%AT	)	:	;
	%EN	%RC														
XM	%TA	%KE	.	.LS	.RS	.A.	.V.	.EV	**	*	/	+	-	=	¬=	>
	>=	<	<=	&	|	.EX	.TH	.EQ	==	...	,	)	:	;	%EN	%RC
DES	.	.LS	.RS	.A.	.V.	.EV	**	*	/	+	-	=	¬=	>	>=	<
	<=	&	|	.EX	.TH	.EQ	==	...	,	)	:	;	%EN	%RC		
X1	.LS	.RS	.A.	.V.	.EV	**	*	/	+	-	=	¬=	>	>=	<	<=
	&	|	.EX	.TH	.EQ	,	)	;	%EN	%RC						
X2	.LS	.RS	.A.	.V.	.EV	**	*	/	+	-	=	¬=	>	>=	<	<=
	&	|	.EX	.TH	.EQ	,	)	;	%EN	%RC						
X3	.A.	.V.	.EV	**	*	/	+	-	=	¬=	>	>=	<	<=	&	|
	.EX	.TH	.EQ	,	)	;	%EN	%RC								
X3U	.A.	.V.	.EV	**	*	/	+	-	=	¬=	>	>=	<	<=	&	|
	.EX	.TH	.EQ	,	)	;	%EN	%RC								
X4	.A.	.V.	.EV	**	*	/	+	-	=	¬=	>	>=	<	<=	&	|
	.EX	.TH	.EQ	,	)	;	%EN	%RC								
X5	.V.	.EV	**	*	/	+	-	=	¬=	>	>=	<	<=	&	|	.EX
	.TH	.EQ	,	)	;	%EN	%RC									
X6	**	*	/	+	-	=	¬=	>	>=	<	<=	&	|	.EX	.TH	.EQ
	,	)	;	%EN	%RC											
X7	*	/	+	-	=	¬=	>	>=	<	<=	&	|	.EX	.TH	.EQ	,
	)	;	%EN	%RC												
X7U	*	/	+	-	=	¬=	>	>=	<	<=	&	|	.EX	.TH	.EQ	,
	)	;	%EN	%RC												
X8	*	/	+	-	=	¬=	>	>=	<	<=	&	|	.EX	.TH	.EQ	,
	)	;	%EN	%RC												
X9	+	-	=	¬=	>	>=	<	<=	&	|	.EX	.TH	.EQ	,	)	;
	%EN	%RC														
XA	=	¬=	>	>=	<	<=	&	|	.EX	.TH	.EQ	,	)	;	%EN	%RC
XB	&	|	.EX	.TH	.EQ	,	)	;	%EN	%RC						
XBU	&	|	.EX	.TH	.EQ	,	)	;	%EN	%RC						
XC	&	|	.EX	.TH	.EQ	,	)	;	%EN	%RC						
XD	|	.EX	.TH	.EQ	,	)	;	%EN	%RC						
XE	.TH	.EQ	,	)	;	%EN	%RC									
XF	.EQ	,	)	;	%EN	%RC										
ASN	,	)	;	%EN	%RC											
STM	)	;	%EN	%RC												
XH	...	,	)	;	%EN	%RC										
XJ	,	)	;	%EN	%RC											
LST	,	)	;	%EN	%RC											
XK	,	%AT	)	;	%EN	%RC										
LSD	,	)	;	%EN	%RC											
PLS	@	%TA	%KE	.	.LS	.RS	.A.	.V.	.EV	**	*	/	+	-	=	¬=
	>	>=	<	<=	&	|	.EX	.TH	.EQ	==	...	,	%AT	)	:	;
	%EN	%RC														
STL	;	%EN	%RC													
PGM																

Figure 18. Left Terminal Delimiters.

LEFT TERMINAL DELIMITERS

NTC	TERMINAL CHARACTERS														
XL	@	%TA	.	.AB	.LS	.RS	.N.	.A.	.V.	.EV	**	%NE	*	/	+	-
	=	¬=	>	>=	<	<=	¬	&	|	.EX	.TH	.EQ	==	...	,	%LI
	%AT	%DE	(	:	%SI	;	%CO	%LC								
IDR	%TA	.	.AB	.LS	.RS	.N.	.A.	.V.	.EV	**	%NE	*	/	+	-	=
	¬=	>	>=	<	<=	¬	&	|	.EX	.TH	.EQ	==	...	,	%LI	%AT
	%DE	(	:	%SI	;	%CO	%LC									
XM	.	.AB	.LS	.RS	.N.	.A.	.V.	.EV	**	%NE	*	/	+	-	=	¬=
	>	>=	<	<=	¬	&	|	.EX	.TH	.EQ	==	...	,	%LI	(	:
	%SI	;	%CO	%LC												
DES	.AB	.LS	.RS	.N.	.A.	.V.	.EV	**	%NE	*	/	+	-	=	¬=	>
	>=	<	<=	¬	&	|	.EX	.TH	.EQ	==	...	,	%LI	(	:	%SI
	;	%CO	%LC													
X1	.AB	.LS	.RS	.N.	.A.	.V.	.EV	**	%NE	*	/	+	-	=	¬=	>
	>=	<	<=	¬	&	|	.EX	.TH	.EQ	==	,	%LI	(	:	%SI	;
	%CO	%LC														
X2	.N.	.A.	.V.	.EV	**	%NE	*	/	+	-	=	¬=	>	>=	<	<=
	¬	&	|	.EX	.TH	.EQ	==	,	%LI	(	:	%SI	;	%CO	%LC	
X3	.N.	.A.	.V.	.EV	**	%NE	*	/	+	-	=	¬=	>	>=	<	<=
	¬	&	|	.EX	.TH	.EQ	==	,	%LI	(	:	%SI	;	%CO	%LC	
X3U	.AB	.LS	.RS	.N.	.A.	.V.	.EV	**	%NE	*	/	+	-	=	¬=	>
	>=	<	<=	¬	&	|	.EX	.TH	.EQ	==	,	%LI	(	:	%SI	;
	%CO	%LC														
X4	.V.	.EV	**	%NE	*	/	+	-	=	¬=	>	>=	<	<=	¬	&
	|	.EX	.TH	.EQ	==	,	%LI	(	:	%SI	;	%CO	%LC			
X5	**	%NE	*	/	+	-	=	¬=	>	>=	<	<=	¬	&	|	.EX
	.TH	.EQ	==	,	%LI	(	:	%SI	;	%CO	%LC					
X6	%NE	*	/	+	-	=	¬=	>	>=	<	<=	¬	&	|	.EX	.TH
	.EQ	==	,	%LI	(	:	%SI	;	%CO	%LC						
X7	%NE	*	/	+	-	=	¬=	>	>=	<	<=	¬	&	|	.EX	.TH
	.EQ	==	,	%LI	(	:	%SI	;	%CO	%LC						
X7U	.AB	.LS	.RS	.N.	.A.	.V.	.EV	**	%NE	*	/	+	-	=	¬=	>
	>=	<	<=	¬	&	|	.EX	.TH	.EQ	==	,	%LI	(	:	%SI	;
	%CO	%LC														
X8	+	-	=	¬=	>	>=	<	<=	¬	&	|	.EX	.TH	.EQ	==	,
	%LI	(	:	%SI	;	%CO	%LC									
X9	=	¬=	>	>=	<	<=	¬	&	|	.EX	.TH	.EQ	==	,	%LI	(
	:	%SI	;	%CO	%LC											
XA	¬	&	|	.EX	.TH	.EQ	==	,	%LI	(	:	%SI	;	%CO	%LC	
XB	¬	&	|	.EX	.TH	.EQ	==	,	%LI	(	:	%SI	;	%CO	%LC	
XBU	.AB	.LS	.RS	.N.	.A.	.V.	.EV	**	%NE	*	/	+	-	=	¬=	>
	>=	<	<=	¬	&	|	.EX	.TH	.EQ	==	,	%LI	(	:	%SI	;
	%CO	%LC														
XC	|	.EX	.TH	.EQ	==	,	%LI	(	:	%SI	;	%CO	%LC			
XD	.TH	.EQ	==	,	%LI	(	:	%SI	;	%CO	%LC					
XE	.EQ	==	,	%LI	(	:	%SI	;	%CO	%LC						
XF	==	,	%LI	(	:	%SI	;	%CO	%LC							
ASN	==	,	%LI	(	:	%SI	;	%CO	%LC							
STM	(	:	%SI	;	%CO	%LC										
XH	,	%LI	(													
XJ	,	%LI	(													
LST	%LI	(														
XK	,	%DE	(													
LSD	%DE	(														
PLS	%LP															
STL	%CO	%LC														
PGM																

Figure 19. Right Terminal Delimiters.

4.2 Interpretation of the Kernel Grammar

The MAD/I language can be described on several levels, each corresponding to a set of definitional rules, many of which can be changed during the course of compilation. At the bottom level a set of lexicographic rules establishes the form of the various identifiers recognized by the compiler. These rules are effective during both the definition and compilation phases of operation, but are modifiable in either case by means of special statements directed to the lexicographic recognizer. Each uniquely named identifier is stored as a separate entry in the symbol table; and, in some cases, identifiers of the same name may be stored separately in a pushdown fashion.

Each symbol table entry corresponding to an identifier is given a set of attributes which establish, among other things, the syntactic class of the entry. The syntactic class of each entry is established either explicitly during the definitional phase or implicitly by default during the compilation phase.. The values assigned each class are chosen according to the interpretation desired, using Figure 9 of Section 4.1 as a guide. By convention, certain types of identifiers are assigned only in certain classes. For instance, identifiers consisting of strings enclosed in primes are usually assigned in the %SIMP, %DECL, %LIST, %COMP, and %END classes (see Figure 9), since these correspond most closely with the notion of statement keyword name. Further-

more, identifiers consisting of strings enclosed in periods are usually assigned in either the unary or binary operator classes, since those correspond most closely with the notion of operator name as popularized in MAD/7090.

A symbol table entry in the translator is created for each new identifier found during the scan of an input program. By default, such an entry is assigned the syntactic class corresponding to operand (%ID). The following lexical structures are identified:

1. Certain characters such as "+" , "-" , and so forth.

2. Certain two-character sequences (digrams) such as "==" , "<=" , and so forth.

3. Names constructed like MAD/7090 variable names, that is strings of letters and numbers, beginning with a letter.

4. Names constructed as in (3) but enclosed in primes.

5. Names constructed as in (3) but enclosed in periods.

6. Constants of various types.

Identifiers of Types 1 and 2 above are usually interpreted as operators and grouping marks. Identifiers of Type 3 above are usually interpreted as variable names which are assigned modes and other attributes in the usual fashion.

Identifiers of Type 4 are usually interpreted as the names of statements and certain constants such as 'TRUE' and 'FALSE.' Finally, identifiers of Type 5 are usually interpreted as names of operators at the expression level.

An operand, either a variable or a constant, is assigned the syntactic class designated by %ID in Figure 9. In general, a parenthesized list may be substituted for an occurrence of an operand anywhere in the language. Since the parsing algorithm used in the compiler does not make use of a table or productions, it is not, in general, possible to restrict the identification of an operand to exclude, say, a parenthesized list or an invalid mode combination where its use would traditionally be ruled invalid. Such distinctions can be made only by the transformation machinery. Within the compiler, an operand is assumed to be represented by a pointer, either actual, as in the case of a register contents, or virtual, as in the case of an intermediary result during expression evaluation.

At any occurrence of an operand in an input program, the operand may be followed by the at sign "@" and a parenthesized list of attribute assignments, which consist of declaration statements. Such a structure is called the attribute notation and may be used in lieu of explicit declarations. Any attribute assigned by the attribute notation is assumed global in scope and no machine computation is generated by its use.

The operations of subscription and of function evaluation are presumed to produce a pointer as a result. The function operation maps an operand on the left and a value on the right into an operand as the result. The value on the right is assumed to be represented as a parenthesized list, perhaps including a sequence of values obtained as the results of a sequence of expressions. For purposes of consistency, the result of a function evaluation is considered an operand, and the code produced by the compiler might well expect a called function to return a pointer to a value rather than the value itself. The subscription operation is considered a special case of the function operation. The subscription operator %TAC is generated as the result of a terminal transformation (see Section 3.1). The component selection operator %KEY is used in much the same sense. Note that the parameter list setup procedure prior to a nested function call is minimized by expecting the function to return a pointer to its result. Furthermore, note that the subscription operation permits the use of indexed machine instructions when singly dimensioned vectors are involved. In the case of multiply dimensioned arrays, a storage mapping function is presumed to map the set of multiple subscripts into a single subscript.

The result of any operation which is assigned a pointer is formally a nonterminal symbol of the kernel grammar and is called a <u>designator</u> (DES) in the productions of Figure 7, Section 4.1. On the other hand, the result of

any expression containing a unary or binary operator or relation is assumed to be a value. A value is distinguished from a designator by the fact that an operation other than address computation is involved and that an intermediate result may be obtained which then must be stored in a temporary location. It is this distinction that rules such expressions as A(B).(C) valid, but (A+B).(C) invalid. The notion of value includes that of operand and may be applied to statements as well as expressions.

The unary and binary operators involved in a particular expression are ranked according to the traditional rules of precedence in the same manner as that popular in 7090 MAD, with an important exception: where in 7090 MAD only one integer is assigned each operator in the ranking, in MAD/I two integers are assigned each operator (see previous section). One reason for this apparent complication is that some operators naturally associate from right to left (e.g., substitution and exponentiation), while other operators naturally associate from left to right (e.g., addition). Thus the definition of new operators within the present hierarchy involves the specification of two precedence "functions" or, alternatively, the specification of one function and a statement as to whether the operator associates from left-to-right or from right-to-left.

Figure 9, Section 4.2, shows all predefined operators in the kernel language and the precedence/class assignments for each. All of those operators in terminal context classes 6 through 8, except the substitution operator, map a value on the left and a value on the right into a value as the result. The substitution operator maps an operand on the left and a value on the right into an operand as the result. With that interpretation, an embedded substitution statement can be used anywhere that an operand is expected, and can lead to some interesting and perhaps useful constructions The result of any expression which is assigned a value is formally a nonterminal symbol of the kernel grammar and is called an assignment (ASN) in the productions of Figure 7. An assignment is also a statement and may be used anywhere that a statement is valid.

The various statements in the language may be organized into several categories on the basis of syntactic type. All of these statements, with a single exception, can be identified by a keyword which is assumed to occur initially. The single exception is the assignment statement and its degeneracies discussed immediately above. Each identifying keyword is assumed a member of an equivalence class identified by one of the metaterminal symbols %SIMP, %COMP, %LIST, %DECL, and %COMP as appearing throughout the succeeding discussions.

A statement of any syntactic type is assumed to be constructed of two units: the prefix, consisting of the identifying keyword followed by a known number of arguments of a known syntactic type, and the scope, consisting of an indefinite number of arguments, all of the same known syntactic type. As used here, the term argument is applied to the nonterminal symbols for designator and assignment and, in addition, others which will be introduced from time to time. Each argument, both in the prefix and in the scope, is separated by such symbols as comma and semicolon, and these features are used in conjunction with context-dependent transformations in the generation of connectives and binding linkages among the arguments.

A macro definition is associated with each statement-identifying keyword in the language. Explicit calls upon the statement-scanning algorithm emitted from such a macro cause the arguments of the prefix to be scanned, and explicit connectives are generated wherever necessary to bind these arguments together and with the scope. No attempt is made during this prefix scan to preserve the natural embedding structure of the kernel language; thus, some rather messy syntactic structures can be defined with a minimum of tricky grammatical specification. It should be emphasized, however, that the entire prefix is interpreted in the kernel grammar as an instance of a metaterminal symbol which is a member of the same equivalence class of which the identifying keyword is a member.

The scope of a statement is a list of arguments separated by commas and semicolons. This list is terminated by either a list separator or statement separator as established by special transformations unique to each statement type. An argument may be one of the following nonterminal symbols, again depending upon statement type:

1. Executable list element (LST) - either an assignment or an instance of the block notation. (The block notation, interpreted to represent a range of elements of a vector or array, is indicated by the " " operator in the same manner as MAD/7090.)

2. Declarative list element (LSD) - either a designator or a special notation developed from the subscript notation and used in connection with certain transformations.

3. Statement (STM) - any of the statements described below and in addition the assignment (ASN).

Using the notions developed here, each of the several statement types can be described in terms of the type of its prefix and the type of list element in its scope.

An <u>assignment statement</u> consists of precisely the assignment itself, which may occur alone as a statement. In the typical case this statement will include the substitution operator "==" and will result in the assignment of a value to a variable. In fact, however, any expression, designator, or even a single identifier can stand alone as a statement. In

the most advanced case of degeneracy the statement is null and no operation is implied except perhaps the assignment of an entry point or statement label (viz., the old CONTINUE statement of MAD/7090). A special transformation is available to detect this condition.

A simple statement consists of a prefix of the %SIMP class and a single STM argument in its scope. A common degeneracy of the simple statement is a statement type consisting only of a prefix. In this case, the scope is null and a special transformation is available to establish the fact.

A compound statement consists of a prefix of the %COMP class followed by a scope of STM arguments separated by semicolons ";" , and terminated by a keyword belonging to the %ENB equivalence class. A particularly useful convention has been adopted in the MAD/I syntax which provides for two forms of the "compound" statement. But here the term "compound" refers to traditional MAD/7090 usage, and not to the more formal nomenclature used here. Of the two "compound" forms, the former, called the compound form of the "compound" statement, consists of a prefix terminated by a semicolon and followed by a scope as described above. The latter, called the simple form of the "compound" statement, consists of a prefix terminated by a comma and followed by a single STM argument. In point of fact, the former is formally a compound statement identified by a keyword of the %COMP class, while the latter is formally a simple statement identified by a keyword of the %SIMP class. Never-

theless, it is possible to make contextual distinctions depending upon the nature of the arguments within the prefix and to reassign the metaterminal symbol class of the statement prefix during the prefix scan, so that one macro definition serves "compound" statements of both forms.

A list statement consists of a prefix of the %LIST class followed by a scope of LST arguments separated by commas. Such statements most often are models of input/output statements in the language. For the purpose of scanning a statement prefix of any class, it is convenient to assign the metaterminal symbol class of the prefix to the %LIST class; and, when the prefix scan is terminated, to reassign the prefix to that class appropriate for the scope scan.

A declarative statement consists of a prefix of the %DECL class followed by a scope of LSD arguments Such statements are most often models of the common declarations in the language The arguments of the scope are constrained to exclude most arithmetic and logical operators; but, in particular, are permitted to contain those operations implied by subscription. Since the subscription operator (%TAG) can occur also in an executable list element, the macro transformation associated with its name must be replaced during the scan of a declarative list element The mechanism for implementing this involves use of a pushdown stack which saves and restores these definitions as necessary.

A program consists of a prefix of the %LC class followed by a scope of STM arguments separated by semicolons ";" and terminated by a keyword of the %RC class. The program represents, of course, the largest structure identified during the compilation process and corresponds to a sentence in the kernel language. According to the usual interpretation, only one statement will occur in the scope, that is, the outermost function definition of the source program. Any additional statements in the scope represent an error condition. Note that the metaterminal symbol %RC can be created explicitly as the result of a transformation or implicitly as the result of an end-of-file condition representing the end of the input text.

V. STRUCTURE OF THE COMPILER

The basic element in any MAD source program is the identifier, used to stand for a program variable, constant, keyword, operator, or punctuation mark. An identifier is extracted from the input text using a set of lexicographic rules which are independent of its membership in these syntactic categories. Using these identifiers as atomic elements, strings representing expressions and statements can be constructed using the set of syntactic rules described in previous sections. Each of these expressions and statements, and ultimately the program itself, has a semantic interpretation rule which assigns to each identified syntactic construct a sequence of machine instructions and

procedure calls. In this connection it is proper to say that the lexicographic recognition rules, the syntactic combinatorial rules, and the semantic interpretation rules are each independent of the others.

The function of the major structural components of the MAD/I compiler parallel this lexical-syntactic-semantic hierarchy. Corresponding to the lexical recognition rules is a processor called ICODE which assembles sequences of input characters into identifiers Corresponding to the syntactic recognition rules is a processor called JSCAN which assembles each sequence of identifiers into a substitution instance of one of the rules of the kernel grammar Associated with each rule of this grammar is a hierarchy of macro definitions which represents the semantic interpretation of the rule. A processor called INTERP interprets these macros and generates calls upon other dependent processors as well. Each macro is written as a sequence of statements of a definitional metalanguage The collection of all those macro definitions which define MAD/I becomes in fact a specification of a dialect of the language; and in this sense, each different collection of macro definitions represents a different dialect of MAD/I.

The principal components of the MAD compiler interconnect as shown in Figure 20. All of these components share a common data structure, or symbol table, into which the source program symbols are coded along with macro definitions, translator variables, and certain intermediate parses Each

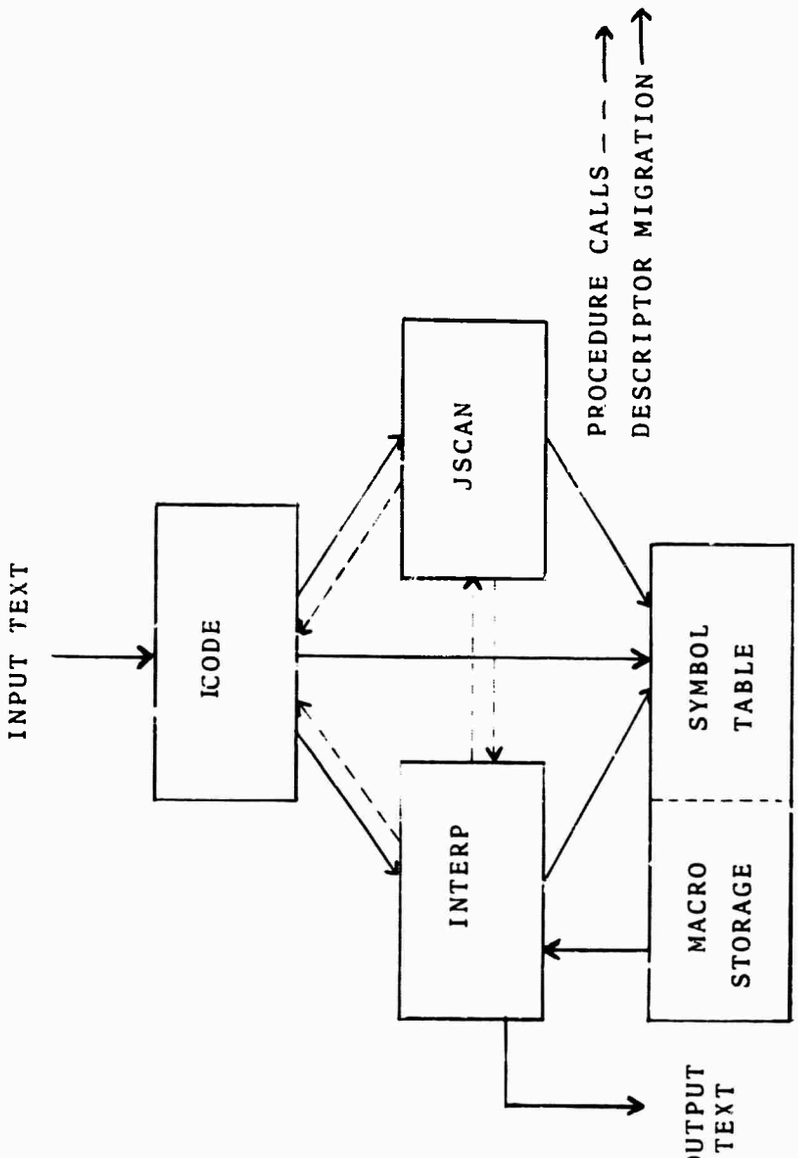


Figure 20. Organization of MAD/I Compiler.

component of the translator and, in principle, a significant fraction of the rather large symbol table can be shared among several concurrent jobs in the timeshared system.

Compilation of a source program requires two passes. In the first pass, each identifier is extracted in turn from the source text, inserted in the symbol table along with default attributes, and assembled by INTERP either into statements of the definitional metalanguage or by JSCAN into sequences of macro calls. During the first pass, attributes are collected and assigned to the various identifiers, and storage allocation information is collected.

During the interlude between the first and second passes, the storage allocation information is processed and the object program storage requirements calculated. Conversion of constants from the external to the internal form is also performed at this time. Finally, the default attributes are assigned all variables and constants which have not been specifically excepted by declaration statements during the first pass.

In the second pass, the macro calls generated during the first pass are expanded and the corresponding object code is generated. Much of the strategy used in the production of the object code from a macro call is established by the macro definition itself, although certain often-used functions such as mode conversions and working register assignments are provided in assembly code rather than in interpreted macro code

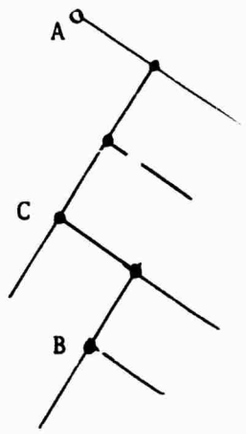
5.1 The Symbol Table

The symbol table is the binding structure through which all the translator components exchange information. Every symbolic variable name expressed in the source program is represented in this table along with the symbolic representation of source program constants. In addition, all those internal symbols used in the various macro definitions are also represented in the symbol table. In particular, certain pre-constructed tables and macro definitions are assumed to be resident in the symbol table before a source program translation can begin. These tables and definitions are created during the definitional phase of translator preparation, and in fact establish the MAD/I language structure.

The symbol table consists of a large binary tree structure, each node of which is represented by a unique symbolic name. In some cases a node may not be in fact named, so in these cases the node is identified by its ordinal position relative to another node. A symbolic representation of nodal position in this tree has been established by ad-hoc convention in the following way: A path from a particular node to its left son is called its link (LNK) while the path to its right son is called its extended attribute pointer (XAP). A path between a node and any of its descendants can be established as the traverse of these links and extended attribute pointers indicated by successive applications of the two operators Ⓛ (link left) and Ⓡ (link right) expressed in postfix form. For example,

A (R)(L)(L)(R)(L) 1.

represents the path from the node named A to the node named B in the tree:



This simple notion can be expressed in a pseudo-algebraic manner in the following way: For each (R) to the right of an expression such as 1, write a set of matching parens in their place. Count the number of (L) terms between an (R) on the left and the next (R) on the right and place this number within the parens representing the (R) on the left. Thus, the expression A(2)(1) stands for the Expression 1. Alternatively, instead of ordinal numeral identification, a node can be identified by name relative to another node. Thus, if C is the name of a node as shown, then the notation A(C)(B) would designate the same node as 1. This pseudo-algebraic notation, called in the

sequel simply the attribute notation, will be used henceforth. This notation is rather more compact than the previous for the kind of structures used here and is considerably easier to parse in definitions.

Each node in the tree-structured symbol table is represented by a block of information including

a the link pointer,

b. the extended attribute pointer,

c a four-entry class-code field (discussed presently),

d. a field containing an interpreted value assigned the node, and

e. the name of the node (possibly null).

The descriptor assigned each such node is the address of its link pointer, so that each link pointer on the left-son chain is in fact a descriptor to the next element on that chain, and each extended attribute pointer on the right-son chain is in fact a descriptor to the next element on that chain.

The class code establishes whether the entry belongs to the class of variables or constants, macro names, or translator variables, and so forth. The values assigned some of these codes are conditioned by the syntactic class to which the symbol belongs (as described previously in Section 2.3); the values of the remaining are chosen arbitrarily. In particular, it is possible to differentiate that symbol table entry which contains a name as a value from that entry which contains a self-defining

(numerical) constant, so that the uses of the remaining fields in connection with the attribute notation do not conflict.

The entries in the class code field are interpreted as a function of the class code value. If the class code designates an operand (e.g., variable name) in the source program, then these entries are interpreted as local attributes; that is, as one-byte attributes which themselves have no dependent attribute structure. These are referenced in the attribute notation as if they were standard symbol table entries with the "value" assigned such entries interpreted as the value of the entry itself. If the class code designates an operator (e.g., a keyword, operator, punctuation or grouping mark) in the source program, then these entries are interpreted as the syntactic type and left- and right-precedence functions respectively.

The value field is designed to, among other things, contain a descriptor. A useful ad-hoc notation designating this value as opposed to the node itself takes the form of subscript notation: VAL(A(2)(1)) identifies that descriptor which is resident in the value field of the node identified by A(2)(1). However, a value field can contain either a name or an ordinal number, and these can be used in attribute notation expressions as well. Thus, A(VAL(B(1))) designates that node found by considering the contents of the value field of B(1) as either the name or the ordinal number of a node on the extended attribute chain of A, depending upon the class code assigned to B(1). Similar notations can be developed for the other local attributes of the entry.

5.2 Lexical Recognizer

All symbolic input to the translator is processed by ICODE, which assembles characters into symbolic names and constant representations. As part of this process, each unique symbol is assigned space in the symbol table along with default attribute assignments. Associated with each such symbol is its descriptor, which is used as its referent in all internal operations. For these purposes, not only the usual variables and constants are considered as symbol table entries, but grouping marks, keywords, and punctuation marks as well. The principal function of ICODE then is to identify each occurrence of a symbol in the input stream and to replace this symbol by its descriptor for use in subsequent operations. The recognition rules used by ICODE in these functions are mostly embedded within the processor itself, although some of these are table-driven and can be set at translation time.

Each identifier used in a source program must be used in only one function. Therefore, obviously, the names assigned to elements in the classes of variables, constants, keywords, operators, and punctuation marks must not overlap. In addition, the names of source program identifiers cannot overlap those used in statement definitions, including those used to define the source language itself. To avoid the naturally occurring conflicts, certain naming conventions have been established which represent sufficient conditions for uniqueness. Thus, variables by convention start with a letter, constants with a

digit, keywords with a prime, and so forth. These conventions are not necessary conditions however, and it is readily possible to assign any identifier to any syntactic class as long as uniqueness criteria are observed.

The lexicographic identification rules used by ICODE can be summarized as follows: The set of available characters is partitioned into subsets of alphabetic, numeric, and special characters Certain special characters stand alone as identifiers, including those normally used as arithmetic operators and punctuation marks Certain other two-character sequences (digrams), including some of the relational operators such as >=, <= and the substitution operator == , stand only in juxtaposition as identifiers. A name consisting of a string of alphabetic and numeric characters, the first of which is an alphabetic character, stands as an identifier and is normally interpreted as a variable name. A name consisting of a string of alphabetic and numeric characters enclosed in either periods "." or primes "'" stands for an identifier. In the former case, such an identifier normally stands for a special operator such as .ABS. , the absolute-value operator; and in the latter case, such an identifier normally stands for a keyword such as 'GO TO' , the name of the MAD/I branch statement

A constant is identified as a string of alphabetic and numeric characters, the first of which is a numeric character. The alphabetic characters in such an identifier are interpreted in such functions as scale factor and radix conversions.

Character constants, that is, those constants that stand for themselves,are delimited by quotation marks """ , between which the quotation mark is identified by two juxtaposed quotation marks in the conventional fashion. Although some default conversion attributes of constants are apparent in their explicit form, no attempt is made in the present MAD/I compiler to convert constants from the external form to the internal representation until the entire source program has been scanned and all global declarations have been collected.

By convention, all symbols used in the definitional process are prefixed with percent signs "%" and all source program symbols are constrained so that the percent sign may not occur first Normally, the translator can be described as being either in the definitional or translational state: in the former, operator and statement macros are defined, and in the latter the source program is translated. A switch is set in ICODE in the definitional state so that

a. all constants are assumed to be of self-defined type, and

b. all symbols beginning with an alphabetic character are prefixed by a percent sign.

Nevertheless, if a constant is prefixed with a special keyword, then it is assumed to be a source program literal; and, if a variable name is prefixed with a percent sign, then the normally occurring percent sign prefixing process is disabled for that symbol

5.3 Syntactic Recognizer

The syntactic recognition algorithm (JSCAN) reads descriptors from an input stream via ICODE and calls upon the macro interpreter (INTERP) with substitution instances of productions as arguments. Some of these productions represent calls on macros which in turn generate connectives—floating addresses and local branches—and may call recursively on JSCAN. Other productions represent calls on macros which have as their primary objective the production of machine code. The first kind of productions may loosely be described as statement macro calls and the second kind as operator macro calls, although the same INTERP machinery is used in both cases.

The term prefiple is used as a generalization of MAD/7090 triple. A prefiple is composed of an operator followed by a list of its operands. The operand list may be of indefinite length, as in the case of the function and subscription operator macro calls, and the operator will be one of those summarized in Figure 9, Section 4.1. Some of the punctuation marks in the source program language are given the class of macro name for the purpose of JSCAN's operations. Two of these currently treated in this manner are the colon and semicolon. In addition, certain operators are invented by terminal transformations detected by JSCAN. Two of these are %NEG and %TAG, neither of which can occur in a source program.

JSCAN is built around an operator-precedence syntax analyzer to which is added a considerable number of contextual transformations. These transformations (see Sections 3.1 and

3.2) are of two classes: one involving an input stream context of two adjacent descriptors, the other involving a context of the two descriptors compared in the precedence relation. Some transformations of each of these two classes are coded in the JSCAN algorithm itself; others are coded as macro definitions.

The classical algorithm (cf. Floyd) which decomposes a stream of text into instances of productions uses a push-down stack in which both terminal and nonterminal characters are temporarily stored. During the analysis, segments of this stack are identified as a production, processed, and deleted by some sort of transformational machinery. In the JSCAN case, each instance of a production is a prefiple and is processed by INTERP. The processing involves the replacement of the production by a nonterminal character which occurs on the left of the equal signs in the table of productions (Figure 7, Section 4.1)

In the particular algorithm used in JSCAN, the nonterminal used to replace a production on the stack is always of operand class, so that, in general, no error checking is possible to differentiate among different productions which involve the same operators. However, if different nonterminals are associated with different mode classes, then the normal mode-context machinery will filter out syntax errors of this type. Note that nowhere in JSCAN itself does the mode of an operand play any part.

The state of JSCAN at any time during compilation of a source program is determined by three descriptions: NXTDSX, LSTDSX, and STKDSX. The first of these, NXTDSX, is the current

descriptor under scan, presumably supplied by ICODE. The second, LSTDSX, is the descriptor read immediately prior to NXTDSX. The third, STKDSX, is the first terminal descriptor found in the stack on a last-in to first-in search.

The pair NXTDSX-LSTDSX represents a context which controls the terminal transformations. Both NXTDSX and LSTDSX are members of equivalence classes of descriptors classified as described in Section 4.2. The class numbers assigned to NXTDSX and LSTDSX are used as coordinates in the terminal context matrix; and the intersection of these coordinates gives access to an integer which is an index in a dispatch table which in turn leads to segments of machine code.

The pair NXTDSX-STKDSX represents the context which drives the precedence algorithm, which is the nucleus of JSCAN. As in the previous case, the class codes of NXTDSX and STKDSX are used as coordinates in the terminal context matrix which gives access to an integer which is an index into a dispatch table. The precedence relations themselves are established by two local attributes of NXTDSX and STKDSX. The terminal context table in this case serves as a convenient handle to invoke precedence transformations.

5.4 The Macro Interpreter

The operation of all the translator processors revolves about INTERP, the internal macro interpreter. This processor interprets commands of a highly stylized definitional metalanguage used to control the decomposition of each source

statement in the program. Some of the commands of this meta-language are used to manipulate symbol table entries, create attribute structures, and assign values of each attribute. Others are used to invoke the other processors of the translator, in particular ICODE and JSCAN. Still others are used within INTERP itself for the control of macro interpetation flow and the defnition of new macros.

The principal definitional structure processed by INTERP is the macro, consisting of from two to half-a-hundred command lines. Each macro is named and may contain instances of parametric substitutions. A macro is invoked when a command of that name is interpreted, and in such a case the parametric substitutions implied are executed. INTERP is so designed that a call can be made by another processor for the purpose of interpreting a single command line passed as an argument. INTERP will interpret this line and then return immediately to the calling processor. If the command involved happens to be a macro name, then that macro is interpreted; and, if the macro contains a call on the calling processor itself, then the whole interpretation process recurses.

A single macro definition is associated with each operator which may occur in a compiler source input expression; and, in addition, other macro transformations may be recursively dependent upon such operator macro definitions. The set of transformations so defined are equivalent in scope to the definition facility built into MAD/7090. Each set of macro

definitions deriving from an expressional operator name corresponds to a define sequence of MAD/7090 deriving from the same operator name. Machinery for the specification of operator precedence and mode context is provided.

In addition to the class of operator macro transformations, a class of keyword macro transformations is included in the resident MAD/I compiler. These macro definitions may call upon the same machinery and pseudo-operation pool as do the operator macro definitions, but, in addition, may call on those pseudo commands which control the operator-precedence grammar-parsing algorithm. A single keyword macro definition is associated with each statement keyword available in the language. The invocation of a keyword macro is in general context dependent, however, and is designed for convenience in the production of connectives, which consist of floating-address assignments and branches.

A command line consists of a macro operator followed by a list of macro operands. Both the operator and each of its operands are represented by descriptor expresssions, consisting of algebraic-like structures in which the various operators are interpreted as operations upon symbol table entries. The result of the interpretation of either an operator or operand is a descriptor, which in turn points to a symbol table entry. If an operator designates a macro name, then the symbol table entry represents a type of line directory which indexes each command line in the macro definition. If the operator

designates a machine instruction name, then the symbol table entry represents an intricately coded driving table for OCODE, a dependent processor which emits the object program byte by byte Finally, if the operator designates a pseudo-command name recognized by INTERP, then the symbol table entry represents a pointer to the processor entry in INTERP itself.

The class of pseudo commands processed by INTERP includes those which create elements of the attribute structure assigned to each symbol table entry and define the value assigned to each of the components of this structure, those which invoke the other translator processors, such as JSCAN and ICODE, those which provide for the definition of new macros, and finally, those which provide a conditional interpretation capability.

Each descriptor of each operand points to a symbol table entry. The symbol table entries may be of any class, and some are allocated and deallocated dynamically by INTERP. Each operand is represented by an expression structured in much the same way as a source program expression, although the operators and punctuation marks have different interpretations. In particular, the attribute notation (see Section 5.1) is subsumed bodily, and the various arithmetic and logical operators are presumed to operate upon the descriptors themselves. Certain special-purpose functions are defined to streamline some of the common operations

REFERENCES

The MAD Manual, Computing Center, The University of Michigan, Ann Arbor, 1967, 130 pp.

Floyd, R.W., "Syntactic Analysis and Operator Precedence," J. ACM, Vol. 10, No. 3, 1963, pp. 316-333.

Floyd, R.W., "Bounded Context Syntactic Analysis," Comm. ACM, Vol. 7, No. 2, 1964, pp. 62-65.

Arden, B.W., and Graham, R.M., "On GAT and the Construction of Translators," Comm. ACM, Vol. 2, No. 7, 1959, pp. 24-26; correction, ibid., No. 11, 1959, pp. 10-11.

Arden, B.W., Galler, B.A., and Graham, R.M., "The Internal Organization of the MAD Translator," Comm. ACM, Vol. 4, No. 1, 1961, pp. 28-31.

Graham, R.M., "Translator Construction," Notes of the Summer Conference on Automatic Programming, Engineering Summer Conferences, The University of Michigan, Ann Arbor, June 1963.

Arden, B.W., Galler, B.A., and Graham, R.M., "An Algorithm for Translating Boolean Expressions," J. ACM, Vol. 9, No. 2, 1962, pp. 222-239.

Bauer, F.L., and Samelson, K., "Sequential Formula Translation," Comm. ACM, Vol. 3, No. 2, 1960, pp. 76-83.

Floyd, R.W., "A Descriptive Language for Symbol Manipulation," J. ACM, Vol. 8, No. 4, 1961, pp. 579-584.

Organick, E.I., A Computer Primer for the MAD Language, Computing Center, The University of Michigan, Ann Arbor, 1961, 183 pp.

Graham, R.M., "Bounded Context Translation," Proceedings of the 1964 Spring Joint Computer Conference, pp. 17-29.

BLANK PAGE

Unclassified

Security Classification

DOCUMENT CONTROL DATA - R&D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)*
THE UNIVERSITY OF MICHIGAN
CONCOMP PROJECT

2a. REPORT SECURITY CLASSIFICATION
Unclassified

2b GROUP

3. REPORT TITLE

THE SYNTACTIC STRUCTURE OF MAD/I

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
Technical Report

5. AUTHOR(S) *(Last name, first name, initial)*

David L. Mills

6. REPORT DATE	7a TOTAL NO. OF PAGES	7b. NO. OF REFS
June 1968	91	

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
DA-49-083 OSA-3050	
b. PROJECT NO.	Technical Report 7
c.	9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*
d.	

10. AVAILABILITY/LIMITATION NOTICES

Qualified requesters may obtain copies of this report from DDC.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
	Advanced Research Projects Agency

13. ABSTRACT

This report describes the formal linguistic structure of MAD/I, an ALGOL-like language proposed for residence in the Michigan Terminal System (MTS). The MAD/I language is designed for general use for all algebraic and many symbol manipulation applications and in particular is designed for extensibility through the definition of new statement structures. This report, presented in a tutorial format, outlines the development of a set of productions which describe the syntax of this language and the derivation of a set of matrices and tables which drive the syntax analysis procedures of the compiler. In particular, a set of syntax transformations is presented which provide a simple but effective means for statement definition. A brief description of the compiler is also given.

DD FORM 1 JAN 64 1473

Unclassified

Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Syntactic Description						
Production System						
Precedence Language						
Context-Free Grammar						
Compiler						
Translator						

INSTRUCTIONS

1. ORIGINATING ACTIVITY: Enter the name and address of the contractor, subcontractor, grantee, Department of Defense activity or other organization *(corporate author)* issuing the report.

2a. REPORT SECURITY CLASSIFICATION: Enter the overall security classification of the report. Indicate whether "Restricted Data" is included. Marking is to be in accordance with appropriate security regulations.

2b. GROUP: Automatic downgrading is specified in DoD Directive 5200.10 and Armed Forces Industrial Manual. Enter the group number. Also, when applicable, show that optional markings have been used for Group 3 and Group 4 as authorized.

3. REPORT TITLE: Enter the complete report title in all capital letters. Titles in all cases should be unclassified. If a meaningful title cannot be selected without classification, show title classification in all capitals in parenthesis immediately following the title.

4. DESCRIPTIVE NOTES: If appropriate, enter the type of report, e.g., interim, progress, summary, annual, or final. Give the inclusive dates when a specific reporting period is covered.

5. AUTHOR(S): Enter the name(s) of author(s) as shown on or in the report. Enter last name, first name, middle initial. If military, show rank and branch of service. The name of the principal author is an absolute minimum requirement.

6. REPORT DATE: Enter the date of the report as day, month, year; or month, year. If more than one date appears on the report, use date of publication.

7a. TOTAL NUMBER OF PAGES: The total page count should follow normal pagination procedures, i.e., enter the number of pages containing information.

7b. NUMBER OF REFERENCES: Enter the total number of references cited in the report.

8a. CONTRACT OR GRANT NUMBER: If appropriate, enter the applicable number of the contract or grant under which the report was written.

8b, 8c, & 8d. PROJECT NUMBER: Enter the appropriate military department identification, such as project number, subproject number, system numbers, task number, etc.

9a. ORIGINATOR'S REPORT NUMBER(S): Enter the official report number by which the document will be identified and controlled by the originating activity. This number must be unique to this report.

9b. OTHER REPORT NUMBER(S): If the report has been assigned any other report numbers *(either by the originator or by the sponsor)*, also enter this number(s).

10. AVAILABILITY/LIMITATION NOTICES: Enter any limitations on further dissemination of the report, other than those imposed by security classification, using standard statements such as:

(1) "Qualified requesters may obtain copies of this report from DDC."

(2) "Foreign announcement and dissemination of this report by DDC is not authorized."

(3) "U. S. Government agencies may obtain copies of this report directly from DDC. Other qualified DDC users shall request through _____."

(4) "U. S. military agencies may obtain copies of this report directly from DDC. Other qualified users shall request through _____."

(5) "All distribution of this report is controlled. Qualified DDC users shall request through _____."

If the report has been furnished to the Office of Technical Services, Department of Commerce, for sale to the public, indicate this fact and enter the price, if known.

11. SUPPLEMENTARY NOTES: Use for additional explanatory notes.

12. SPONSORING MILITARY ACTIVITY: Enter the name of the departmental project office or laboratory sponsoring *(paying for)* the research and development. Include address.

13. ABSTRACT: Enter an abstract giving a brief and factual summary of the document indicative of the report, even though it may also appear elsewhere in the body of the technical report. If additional space is required, a continuation sheet shall be attached.

It is highly desirable that the abstract of classified reports be unclassified. Each paragraph of the abstract shall end with an indication of the military security classification of the information in the paragraph, represented as *(TS), (S), (C), or (U)*.

There is no limitation on the length of the abstract. However, the suggested length is from 150 to 225 words.

14. KEY WORDS: Key words are technically meaningful terms or short phrases that characterize a report and may be used as index entries for cataloging the report. Key words must be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location, may be used as key words but will be followed by an indication of technical context. The assignment of links, rules, and weights is optional.